VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2020 Bc. Peter Cíbik

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

PODPORA KRYPTOGRAFICKÝCH PRIMITIV V JAZYCE P4
P4 CRYPTOGRAPHIC PRIMITIVE SUPPORT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Peter Cíbik**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. David Smékal**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Peter Cíbik ***ID:*** 186705
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Podpora kryptografických primitiv v jazyce P4

POKYNY PRO VYPRACOVÁNÍ:

V rámci diplomové práce se seznamte s jazykem P4_16 a s jeho podporou externích bloků. Dále se seznamte s prostředky pro mapování jazyka P4_16 na čipy FPGA. Navrhněte a implementujte podporu vybraného kryptografického externího bloku jazyka P4_16 na vybrané platformě. Použijte existující IP Core hašovací funkce SHA-3. Ověřte funkčnost a výkonové parametry výsledné implementace. V závěru diskutujte dosažené výsledky a možnosti rozšíření práce.

DOPORUČENÁ LITERATURA:

[1] P4 Language Consortium, 2018. P4_16 Language Specification [online]. 1.1.0. Dostupné z: https://bit.ly/2lJ9Db6

[2] BOSSHART, Pat, a spol. P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 2014, 44.3: 87-95.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. David Smékal
Konzultant: Ing. Denis Matoušek (Netcope Technologies, a.s.)

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Táto diplomová práca sa zaoberá problematikou bezpečnosti vysoko-rýchlostnej komunikácie, čo vedie k použitie hardvérovvých akcelerátorov, v tomto prípade vysoko-rýchlostných sieťových kariet s FPGA čipom. Venuje sa zjednodušeniu samotného vývoja aplikácii pre FPGA akcelerátory pomocou kompilátoru P4 do VHDL. Popisuje rozšírenie kompilátoru o podporu kryptografických externých objektov. V úvode sa venuje teoretickému základu jazyka P4 a použitým technológiám. Popisuje mapovanie externých objektov do P4 zreťazenia a teda na FPGA čip. Následne sa venuje kryptografickému externému objektu, vytvoreniu kompatibilnej obálky a verifikácii návrhu. V závere popisuje samotnú implementáciu a rozšírenie kompilátoru, podporu kryptografického externého objektu a zhodnocuje dosiahnuté výsledky.

KĽÚČOVÉ SLOVÁ

FPGA, VHDL, Netcope P4, P4, Netcope, kryptografia, HASH, SHA–3

ABSTRACT

This diploma thesis deals with the problem of high-speed communication security which leads to the usage of hardware accelerators, in this case high-speed FPGA NICs. Work with simplification of development of FPGA hardware accelerator applications using the P4 to VHDL compiler. Describes extension of compiler of cryptographic external objects support. Teoretical introduction of the thesis explains basics of P4 language and used technologies. Describes mapping of external objects to P4 pipeline and therefore to FPGA chip. Subsequently deals with cryptographic external object, compatible wrapper implementation and verification of design. Last part describes implementation and compiler extension, cryptographic external object support and summarizes reached goals.

KEYWORDS

FPGA, VHDL, Netcope P4, P4, Netcope, cryptography, HASH, SHA–3

CÍBIK, Peter. *Podpora kryptografických primitiv v jazyce P4*. Brno, 2020, 101 s. Diplomová práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedúci práce: Ing. David Smékal

VYHLÁSENIE

Vyhlasujem, že som svoju diplomovú prácu na tému „Podpora kryptografických primitiv v jazyce P4“ vypracoval samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej diplomovej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi Ing. Davidovi Smékalovi a odbornému konzultantovi spoločnosti Netcope Technologies, a.s. pánovi Ing. Denisovi Matouškovi za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Brno ....

podpis autora

Obsah

Úvod

V súčasnosti je sieťová komunikácia vecou, bez ktorej by sme si nevedeli predstaviť bežný život. Neustály technologický rozvoj prináša nové a nové zariadenia, ktoré medzi sebou komunikujú, čím neustále navyšujú objem spracovávaných dát a požadované rýchlosti ich spracovania. Stále teda narastajú aj nároky na sieť ako takú a na sieťové prvky, ktoré komunikáciu obsluhujú alebo spracovávajú. Vzhľadom na vysoké prenosové rýchlosti, ktoré je nutné dosahovať v hlavných častiach sietí sa v poslednej dobe dostávajú do popredia akcelerátory. Ide o podporné prvky, ktorým je dedikovaná určitá činnosť, ktorú štandardné procesory a zariadenia nedokážu vykonávať v požadovanej záťaži. Hovoríme napríklad o vysoko-rýchlostných sieťových kartách osadených FPGA čipom, ktoré dokážu spracovávať sieťovú prevádzku.

Vývoj aplikácii pre tieto akcelerátory je často náročná činnosť či už z časového alebo znalostného hľadiska, keďže sa jedná o vývoj pre FPGA čip. Naskytá sa teda myšlienka tento vývoj uľahčiť pomocou vysoko-úrovňového jazyka P4. Jedná sa o jazyk na popis spracovávania sieťovej prevádzky a konfiguráciu jednotlivých prvkov siete. Preto podpora konfigurácie akcelerátoru v tomto jazyku podporuje myšlienku určitého uceleného riešenia, kedy je možné rôzne sieťové prvky konfigurovať iba pomocou tohto jazyka, čím značne uľahčuje znalostnú a časovú bázu vývoja konkrétnych riešení.

Ďalšou nutnosťou dnešnej doby je zabezpečenie tejto vysoko-rýchlostnej sieťovej komunikácie. Ide o samotné zabezpečenie dôvernosti dát pomocou šifrovania, integrity pomocou otlačku správy hashovacou funkciou, prípadne autentičnosti pomocou digitálneho podpisu.

Predstavenie práce

Táto diplomová práca sa zaoberá skĺbením problematiky vysoko-rýchlostnej sieťovej komunikácie, vývoja akceleračných nástrojov, konkrétne vývoja pre vysoko-rýchlostné sieťové karty osadené FPGA čipom a zabezpečenia prenášaných dát. Cieľom práce je zoznámiť sa s revíziou jazyka $P4_{16}$ a jej podporou externých objektov. Ďalej sa zoznámiť s prostriedkami mapovania externých objektov na FPGA čip. Navrhnúť implementáciu kryptografických externých objektov do P4 zreťazenia a ich podporu do prvkov jazyka P4. Následne vytvoriť simuláciu zapojenia vybraného kryptografického externého objektu a jeho využitia v P4 zreťazení a overiť teda funkcionalitu daného riešenia. Otestované riešenie implementovať ako rozšírenie kompilátoru a implementovať podporu konkrétneho kryptografického externého objektu s použitím existujúcej implementácie SHA3 hashovacej funkcie. Prakticky

overiť celé rozšírenie a podporu kryptografického externého objektu a zhodnotiť výsledky.

Štruktúra práce

Práca je rozdelená do určitých logických celkov. Na začiatku spracováva problematiku samotného jazyka P4 a jeho podporu externých objektov. Ďalej popisuje jednotlivé technológie použité najmä v praktickej časti. Následne popisuje problematiku a návrh podpory externých objektov a ich mapovania do P4 zreťazenia a FPGA čipu. Popisuje externý objekt, vytvorenie obálky pre kompatibilitu rozhrania a otestovanie návrhu. Posledná časť sa zaoberá samotným rozšírením kompilátoru, pridaním podpory kryptografického externého objektu, overením celého riešenia a zhodnotením výsledkov. Jednotlivé časti detailnejšie popisujú:

- **Jazyk P4** (kapitola 1) – popisuje samotný jazyk P4, porovnáva jednotlivé revízie a následne popisuje jeho prvky a podporu externých objektov.
- **Použité technológie** (kapitola 2) – popisuje jednotlivé technológie použité najmä v praktickej časti tejto práce.
- **Mapovanie externých objektov do P4 zreťazenia** (kapitola 3) – rozoberá problematiku typu externých objektov a navrhuje riešenia ich samotného mapovania do P4 zreťazenia respektíve na platformu FPGA.
- **P4 zreťazenie s externým objektom** (kapitola 4) – popisuje jednotlivé kroky potrebné na vytvorenie verifikácie a overenie funkčnosti návrhu podpory externého objektu, hashovacej funkcie v konkrétnom použití a integrácii v P4 zreťazení.
- **Podpora kryptografických externých objektov** (kapitola 5) – popisuje samotné rozšírenie kompilátoru, podporu konkrétneho kryptografického externého objektu a overenie riešenia.

1 Jazyk P4

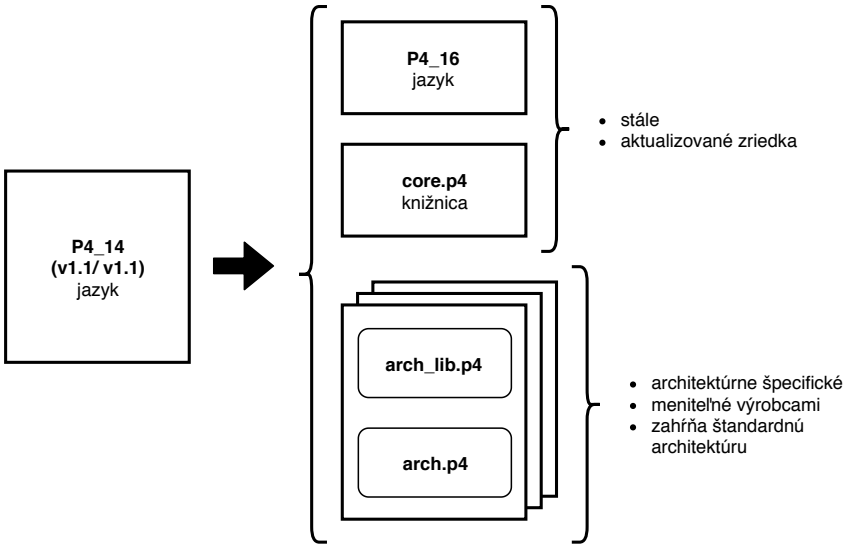
Táto kapitola popisuje jazyk P4 ako taký, rozdiely medzi verziou 14 a 16 a následne externý objekt jazyka $P4_{16}$.

Názov P4 vznikol na základe názvu článku a určenia, pre ktoré bol jazyk P4 vyvinutý – Programming Protocol-independent Packet Processors [1]. Jedná sa o vysokoúrovňový jazyk pre programovanie protokolne-nezávislého spracovania paketov. Hlavné tri výhody jazyka P4 môžeme popísať ako [1]:

- **rekonfigurovateľnosť** – je možné zmeniť spôsob spracovania paketov po nasadení systému,
- **protokolová nezávislosť** – zariadenia nie sú viazané na špecifické protokoly,
- **platformná nezávislosť** – spracovanie paketov je možné popísať nezávisle na cieľovej platforme.

1.1 $P4_{14}$ versus $P4_{16}$

$P4_{16}$ verzie 1.2.0 ako nová revízia jazyka P4 prináša spätne-nekompatibilné zmeny v syntaxi a sémantike samotného jazyka. Popis vývoja a porovnanie jednotlivých revízií jazyka popisuje obrázok 1.1. Je možné vidieť, že veľa prvkov (angl.: features) bolo odobratých zo samotného jazyka a presunutých do knižníc. Napríklad okresaním počtu kľúčových slov (angl.: keywords) zo sedemdesiat na menej než štyridsať, sprevádzaných knižnicou fundamentálnych konštrukcií potrebných na väčšinu P4 funkcií [2].

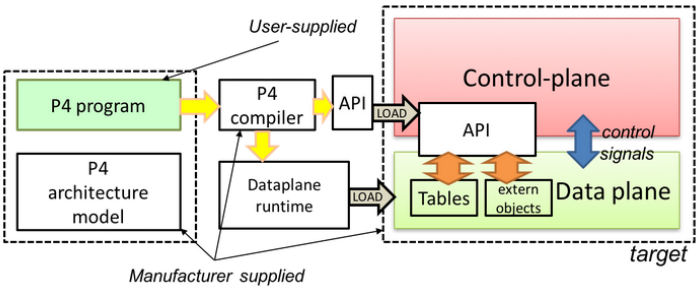


Obr. 1.1: Porovnanie jazyka $P4_{14}$ a $P4_{16}$ [2]

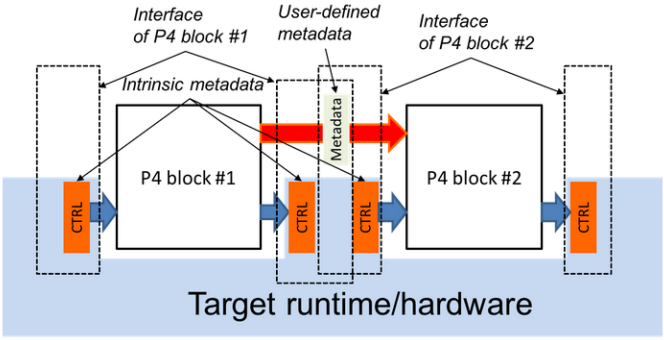
V náväznosti na knižnice a odstránenie určitých prvkov, zavádza $P4_{16}$ konštrukciu jazyka zvanú `extern` (externý objekt). Umožňuje definovať externé objekty nezávislé na samotných prvkoch jazyka P4, a rozšíriť tak funkcionalitu. Keďže veľa prvkov bolo transformovaných do elementov knižníc, napríklad čítač (angl.: counter), musia byt definované ako externé objekty. Externé objekty ako také podrobnejšie popisuje sekcia 1.3. Jednou z ďalších rozšírených možností, ktorú prináša verzia jazyka $P4_{16}$ je konfigurovateľný deparser (skladač paketu) [2].

1.2 $P4_{16}$

Najnovšia verzia jazyka P4, $P4_{16}$, definuje typický postup (angl.: workflow) ako programovať zariadenie pomocou jazyka P4 ako je zobrazené na obrázku 1.2. Ak je v jednom zariadení viac blokov konfigurovateľných pomocou jazyka P4, špecifikácia definuje ich rozhranie ako je možné vidieť na obrázku 1.3. Každý blok môže byt definovaný separátnou časťou zdrojového kódu programu P4. Výrobca zariadenia definuje architektúrny popis a kompilátor pre danú platformu. P4 programátor následne definuje P4 program pre danú architektúru, pričom môže využiť všetky tieto konštrukcie.



Obr. 1.2: Schéma programovania cieľového zariadenia pomocou jazyka P4 [2]



Obr. 1.3: Rozhranie P4 programu [2]

Následne definuje jednotlivé prvky abstrakcie, ktoré ako jadro jazyka P4 môžeme popísať nasledovne:

- **Header types** – popisuje formát hlavičky v rámci paketu a teda jednotlivé políčka a ich veľkosť.
- **Parser** – definuje povolené sekvencie hlavičiek, jednotlivé hlavičky, ako ich identifikovať a extrahovať.
- **Tables** – asociujú užívateľom-definované kľúče k jednotlivým akciám.
- **Actions** – časti kódu popisujúce, ako sú jednotlivé hlavičky a metadáta spracované a ako je s nimi zachádzané.
- **Match-action units** – vytvárajú kľúče z políčok hlavičiek a metadát, invokujú vyhľadanie akcie na základe kľúča v tabuľke a následne spustia vybranú akciu.
- **Control flow** – program popisujúci spracovanie paketov na vybranej platforme, taktiež definuje `Deparser`.
- **Extern objects** – architektúrne špecifické externé objekty bližšie popísané v časti 1.3.
- **User-defined metadata** – užívateľom-definované metadáta asociované s každým paketom.
- **Intrinsic metadata** – architektúrne-závislé metadáta asociované každému paketu.

Hlavné a najzaujímavejšie z nich sú podrobnejšie definované ako:

Header types

Definuje hlavičky, ich jednotlivé políčka a ich dĺžku. Je deklarovaný ako typ `header`.

Výpis 1.1 následne uvádza príklad deklarácie typu hlavičiek `Ethernet_h`, ako hlavičku protokolu ethernet a `IPv4_h` ako hlavičku pre protokol IP verzie 4 s použitím variabilnej dĺžky políčka.

Výpis 1.1: Deklarácia hlavičiek pre protokol ethernet a IP verzie 4 [2]

```
1 header Ethernet_h {
2    bit<48> dstAddr;
3    bit<48> srcAddr;
4    bit<16> etherType;
5 }
6
7 header IPv4_h {
8    bit<4>       version;
9    bit<4>       ihl;
```

```
10      bit<8>        diffserv;
11      bit<16>       totalLen;
12      bit<16>       identification;
13      bit<3>        flags;
14      bit<13>       fragOffset;
15      bit<8>        ttl;
16      bit<8>        protocol;
17      bit<16>       hdrChecksum;
18      bit<32>       srcAddr;
19      bit<32>       dstAddr;
20      varbit<320>   options;
21  }
```

Jednotlivé definované typy hlavičiek sú následne používané a spracovávané v celom P4 programe.

Parser

Je definovaný ako konečný stavový automat. Počiatočný stav je vždy `state start`, z ktorého následne prechádza do jednotlivých ďalších možných stavov. Popisuje možnú náväznosť jednotlivých hlavičiek pre ich následné spracovanie. Stará sa o extrahovanie jednotlivých bitov paketu do políčok hlavičiek. Uvádza sa kľúčovým slovom `parser`. Pracuje na základe jednotlivých stavov spracovania a prechodou medzi nimi. Jednoduchý parser pre protokol MPLS je zobrazený vo výpise 1.2.

Výpis 1.2: Deklarácia časti parser pre protokol MPLS [2]

```
1  parser P(packet_in b, out Pkthdr p) {
2      state start {
3          b.extract(p.ethernet);
4          transition select(p.ethernet.etherType) {
5             0x8847: parse_mpls;
6             0x0800: parse_ipv4;
7          }
8      }
9      state parse_mpls {
10          b.extract(p.mpls.next);
11          transition select(p.mpls.last.bos) {
12             0: parse_mpls; // This creates a~loop
13             1: parse_ipv4;
```

```
14          }
15      }
16      // other states omitted
17  }
```

Control Blocks

Spracováva dáta extrahované blokom Parser. Control blok je deklarovaný pomocou mena, parametrov a následne deklaráciami konštánt, Table a Action častí a premenných. Všeobecná deklarácia je zobrazená vo výpise 1.3.

Výpis 1.3: Deklarácia control bloku [2]

```
1  controlDeclaration
2      : controlTypeDeclaration optConstructorParameters
3        /* controlTypeDeclaration cannot contain type
4        parameters */
5        '{' controlLocalDeclarations APPLY controlBody '}'
6      ;
7
8  controlLocalDeclarations
9      : /* empty */
10     | controlLocalDeclarations controlLocalDeclaration
11     ;
12
13 controlLocalDeclaration
14     : constantDeclaration
15     | variableDeclaration
16     | actionDeclaration
17     | tableDeclaration
18     | instantiation
19     ;
20
21 controlBody
22     : blockStatement
23     ;
```

Actions

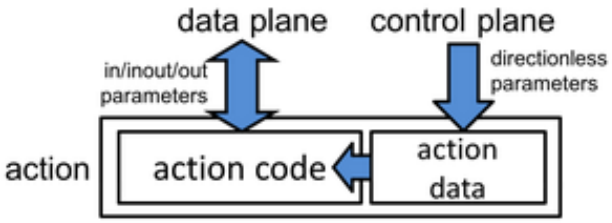
Fragment zdrojového kódu, ktorý môže čítať a zapisovať do spracovávaných dát. Pomocou akcií je možné zasahovať do spracovávaných dát alebo na ich základe dáta konkrétne spracovať. Pre akciu je vyhradené kľúčové slovo `action`. Obrázok 1.4 zobrazuje abstraktný model akcie a výpis 1.4 a 1.5 zase jej deklaráciu.

Výpis 1.4: Deklarácia bloku action [2]

```
1 actionDeclaration
2     : optAnnotations ACTION name '(' parameterList ')' /
3     blockStatement
4     ;
```

Výpis 1.5: Deklarácia bloku action pre akciu Forward_a [2]

```
1 action Forward_a(out bit<9> outputPort, bit<9> port) {
2     outputPort = port;
3 }
```



Obr. 1.4: Abstraktný model action bloku [2]

Tables

Kľúčovým slovom `table` je uvádzaná tabuľka popisujúca `match-action` blok, a teda blok takzvanej zhody a akcie. Deklarácia obsahuje samotnú deklaráciu tabuľky a najčastejšie akcií (Action) a kľúčou (key). Všeobecnú deklaráciu je možné vidieť vo výpise 1.6 nasledovaný výpisom 1.7, kde je zobrazená deklarácia tabuľky Fwd používajúcej dva kľúče.

Výpis 1.6: Deklarácia table bloku [2]

```
tableDeclaration
    : optAnnotations TABLE name '{' tablePropertyList '}'
    ;

tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY '=' '{' keyElementList '}'
    | ACTIONS '=' '{' actionList '}'
    /* immutable entries */
    | CONST ENTRIES '=' '{' entriesList '}'
    | optAnnotations CONST nonTableKwName '= /
    ' initializer ';'
    | optAnnotations nonTableKwName '=' initializer ';'
    ;

nonTableKwName
   : IDENTIFIER
   | TYPE_IDENTIFIER
   | APPLY
   | STATE
   | TYPE
   ;
```

Výpis 1.7: Deklarácia table bloku tabuľky Fwd [2]

```
table Fwd {
    key = {
       ipv4header.dstAddress : ternary;
       ipv4header.version    : exact;
    }
    ...
}
```

Knižnica core.p4 definuje tri druhy políčka `match_kind` a to:

```
match_kind {
    exact,
    ternary,
    lpm
}
```

, kde jednotlivé typy korešpondujú so špecifikáciou jazyka $P4_{14}$[1]:

- **exact** – vstupná hodnota je porovnaná s tabuľkou a musia sa presne zhodovať
- **ternary** – pred vykonaním porovnávania je uplatnená maska pomocou operácie AND so vstupnou hodnotou, ktorá uplatní porovnávanie iba na určité bity vstupnej hodnoty a následne sa porovná s tabuľkou
- **lpm** – jedná sa o špeciálny typ ternary zhody, kedy sa uplatňuje prefix na základe vstupnej masky

Deparser

Stará sa o zloženie paketu z jednotlivých hlavičiek a iných častí. Je konfigurovateľný, a teda nemusí skladať paket rovnako, ako bol na začiatku rozdelený v časti Parser. Jazyk P4 nedefinuje kľúčové slovo pre tento blok, a tak je popísaný ako blok control. Výpis 1.8 zobrazuje konfiguráciu deparser bloku, ktorá zapíše ako prvé ethernet hlavičku a následne IP hlavičku do `packet_out` [2].

Výpis 1.8: Deparser blok zapisujúci hlavičky do packet_out [2]

```
1 control TopDeparser(inout Parsed_packet p, packet_out b) {
2     apply {
3         b.emit(p.ethernet);
4         b.emit(p.ip);
5     }
6 }
```

Detailnejší popis, príklady použitia alebo iné potrebné veci je možné čerpať z oficiálnej špecifikácie jazyka $P4_{16}$ [2].

1.3 Externé objekty $P4_{16}$

Externý objekt (extern) jazyka je konštrukcia sprístupňujúca komponenty cieľovej architektúry pre použitie v jazyku P4. Kedže sú platformne závislé, je nutné ich

[1] https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf

dodávať ako knižnice spolu s kompilátorom P4 pre danú platformu. Samotný extern popisuje rozhranie, ktoré objekt poskytuje pre data plane[2]. Popisuje množinu funkcií, ktoré sú implementované objektom, nie však implementáciu samotnú [2]. Príklad uvedený vo výpise 1.9 popisuje operácie poskytované externým objektom na výpočet kontrolného súčtu (angl.: checksum) [2].

Výpis 1.9: Extern popisujúci objekt na výpočet kontrolného súčtu [2]

```
1  extern Checksum16 {
2      Checksum16();
3      // prepare unit for computation
4      void clear();
5      // add data to checksum
6      void update<T>(in T data);
7      // remove data from existing checksum
8      void remove<T>(in T data);
9      // get the checksum for the data added since
10     // last clear
11     bit<16> get();
12 }
```

Špecifikácia popisuje deklaráciu externého objektu takto:

Výpis 1.10: Deklarácia externého objektu [2]

```
1  externDeclaration
2      : optAnnotations EXTERN nonTypeName optTypeParameters
3          /'{' methodPrototypes '}'
4      | optAnnotations EXTERN functionPrototype ';'
5      ;
```

Deklaruje objekt a všetky metódy, ktoré môžu byť volané. Môžu byť špecifické pre danú platformu. Knižnica P4 Core deklaruje preddefinované externé objekty `packet_in` a `packet_out`. Postupným vývojom sa uvažuje o vytvorení množiny štandardizovaných externých objektov, ktoré budú neskôr prvkom samotného jazyka P4 a jeho Core knižnice, za účelom čoho vzniká napríklad špecifikácia[3] PSA (Portable Switch Architecture) [2].

[2]skupina algoritmov popisujúca transformácie na paketoch vykonávané pomocou systému spracovania paketov (angl.: packet-processing system) [2]

[3]`https://p4.org/p4-spec/docs/PSA-v1.1.0.html#sec-psa-externs`

2 Použité technológie

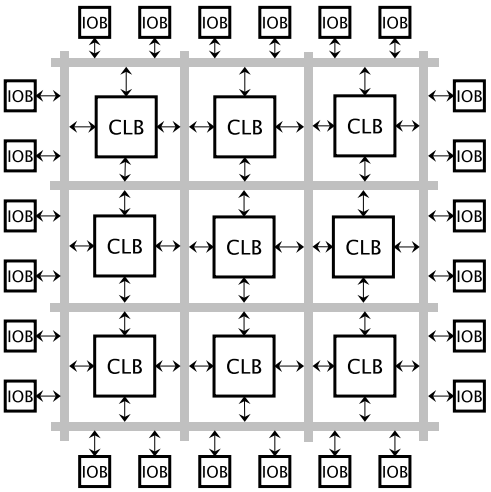
V tejto kapitole sú predstavené použité technológie obsiahnuté v samotnom riešení. Stručne popisuje programovateľné hradlové polia (angl.: FPGA – Field Programable Gate Array), uvádza do základov jazyka VHDL, zoznamuje čitateľa s produktom Netcope P4 a následne popisuje sieťové karty osadené FPGA čipom.

2.1 Programovateľné hradlové pole

Programovateľné hradlové pole je univerzálny programovateľný logický obvod. Vyznačuje sa možnosťou zmeny jeho vnútorného zapojenia – funkcionality a teda jeho flexibilitou. Vnútorné zapojenie obvodu, a teda jeho správanie, je možné konfigurovať jedným z popisných jazykov na hardvér (angl.: HDL – Hardware Description Language) a pomocou syntetizačných a implementačných nástrojov následne mapovať na čip.

Flexibilita a hlavne finančná úspora FPGA obvodov oproti ASIC (Application Specific Integrated Circuit) obvodom je enormná, keďže vývoj ASIC obvodu, ktorý má vždy iba konkrétne využitie a teda nie je flexibilný, zaberá násobne dlhší čas a finančné náklady na výrobu sú taktiež niekoľkokrát väčšie. Preto sú čoraz populárnejšie v akceleračných nástrojoch.

Obrázok 2.1 zobrazuje obecnú štruktúru FPGA čipov skladajúcu sa z programovateľných logických blokov (angl.: CLB – Configurable Logic Block), vstupne-výstupných blokov (angl.: IOB – Input-Output Block) a prepojovacej štruktúry. K obecnej štruktúre potom jednotlivé FPGA čipy zahŕňajú rôzne, špecializované bloky.



Obr. 2.1: Základná štruktúra FPGA [3, 4]

Programovateľný logický blok

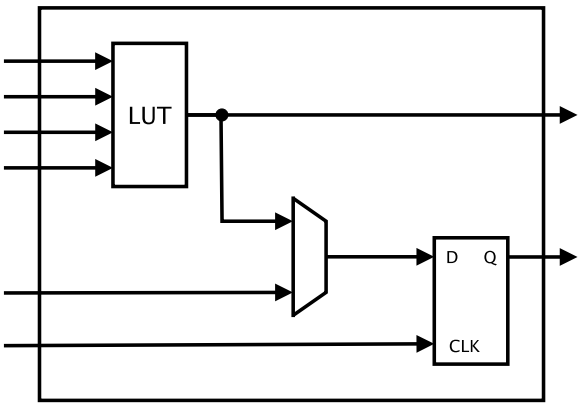
Typicky sa programovateľný logický blok skladá, ako je možné vidieť na obrázku 2.2, z generátoru logickej funkcie, ktorým je vyhľadávacia tabuľka (angl.: LUT – Look-up Table), multiplexoru a klopného obvodu. Ďalej obsahuje lokálne prepojovacie pole týchto prvkov.

Vyhľadávacia tabuľka realizuje kombinačné funkcie. Jedná sa v podstate o malú pamäť typu RAM (Random Access Memory), ktorej obsah je určený pomocou konfiguračných dát.

Multiplexory umožnujú prepojit v danú funkciu na dalšie bloky v rovnakom reze, a tým vytvoriť funkciu väčšej šírky ako je šírka jednej vyhľadávacej tabuľky.

Klopný obvod sa využíva na realizáciu sekvenčných logických funkcií, pričom využíva synchrónne alebo asynchrónne riadenie. Prevažne sa využíva klopný obvod typu D.

Využitie jednotlivých prvkov môže byť nezávislé, k čomu slúžia prídavné vstupy a výstupy a lokálna prepojovacia štruktúra či multiplexory [4, 5, 6].



Obr. 2.2: Schéma programovateľného logického bloku [3, 6]

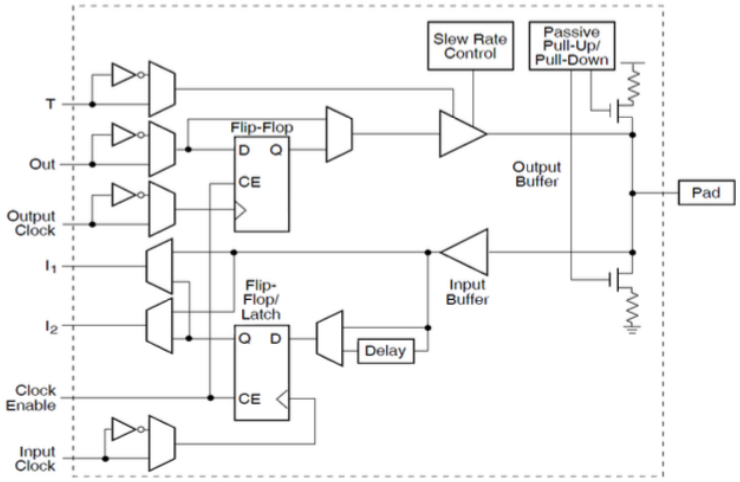
Programovateľný vstupne-výstupný blok

Využívajú sa na prepojenie FPGA čipu s jeho okolím. Každý vstup respektíve výstup FPGA čipu je realizovaný pomocou týchto blokov. Najčastejšie pozostávajú zo:

- vstupné a výstupné zosilovače (angl.: IBUF – Input BUFfer, OBUF – Output BUFfer),
- vstupné a výstupné registre,
- programovateľný invertor,
- väzba na susedné bloky - podpora diferenčných štandardov,
- impedančné prispôsobenie,
- ochranný obvod,

- oneskorovací člen,

ako je možné vidieť na obrázku 2.3 [4, 6].



Obr. 2.3: Obecná schéma zjednodušeného vstupne-výstupného bloku [6]

Programovateľná prepojovacia štruktúra

Jedná sa o prepojovaciu štruktúru slúžiacu k vzájomnému prepojeniu jednotlivých vzdialených logických blokov a k prepojeniu so vstupne-výstupnými blokmi. Je rozmiestnená rovnomerne po celej ploche čipu vo forme matice [4, 5].

2.2 HDL jazyky

V tejto sekcii sú popísané HDL jazyky, jazyky na popis hardvéru, ktoré sú využívané v tejto práci.

Jedná sa o jazyky popisujúce štruktúru zapojenia a správanie sa číslicových obvodov v textovej podobe.

VHDL – .vhd

Jedná sa o jazyk najviac rozšírený v Európe. Používa sa na popis hardvéru, prípadne na popis simulácií. Výpis 2.1 znázorňuje základnú štruktúru zdrojového kódu jazyka VHDL. Popis teda pozostáva vždy z časti `entity` popisujúcej rozhranie daného modulu a následne z `architektúry`, ktorá popisuje samotnú funkcionalitu a správanie modulu. Taktiež je možné využívať knižnice a knižné balíčky, a teda funkcie a veci nimi definované. Najznámejšie z nich sú `ieee.std_logic_1164` a `ieee.numeric_std` Jazyk ďalej podporuje ako paralelné, tak sekvenčné príkazy [5, 6, 7]. Podrobnejší popis je možné nájsť v mojej bakalárskej práci [3].

Výpis 2.1: Zakladná schéma zdrojového súboru jazyka VHDL [5]

```
1  -- [] - volitelne pouzitie
2  -- {} - mozne pouzit opakovane
3  -- | - alternativa
4  -- () - zoskupenie
5  -- ID - identifikator
6
7  library Meno_Kniznice [{, Meno_Kniznice}];
8  use Meno_Kniznice.Meno_Knizneho_Balicku.[all |
9    identifikator_alebo_operator];
10
11 entity NazovEntity is
12   [generic ({ID : TYPEID [:= vyraz];});]
13   [port({ID: [in | out | inout | buffer]
14     TYPEID [:= vyraz];});]
15   [{deklarace}]
16 [begin
17   {paralelne_prikazy}]
18 end [entity] NazovEntity;
19
20 architecture Meno_Architektury_/_ID of Meno_Entity is
21   [{definicie_a_deklaracie}]
22 begin
23   [{paralelne_prikazy}]
24 end [architecture] [Meno_Architektury_/_ID];
```

SystemVerilog – .sv

Okrem toho, že je jedným z HDL jazykov patrí aj do rodiny HVL (Hardware Verification Language) jazykov. Používa sa teda najmä na verifikovanie hardvéru. Tým, že je najnovší z vyššie popísaných a je rozšírením samotného jazyka Verilog, má v sebe prvky rôznych moderných jazykov zľahčujúcich prácu. Medzi prvky zvyšujúce efektivitu verifikácie patrí [8, 9]:

- **objektovo orientovaný** (angl.: OOP – Object-Oriented Programming) – podporuje a uľahčuje vývoj za použitia spoločných návrhových vzorov a viackrát použiteľných komponent. Využíva mechanizmy zapúzdrenia, dedenia a polymorfizmus,
- **obmedzené náhodné generovanie stimulov** (angl.: constrained-random

stimulus generation) – náhodné generovanie vstupných stimulov, a teda pokrytie veľkého množstva možností. Na zachovanie správnosti vstupov sa dá použiť obmedzenie (angl.: constrain) pre náhodné generovanie,

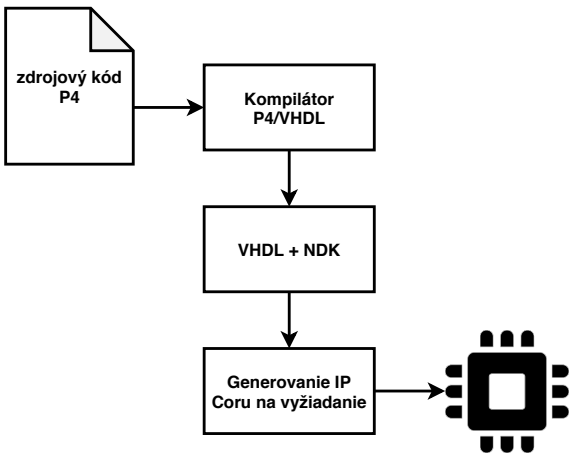
- **verifikácia postavená na výrokoch** (angl.: ABV – Assertion-Based Verification) – používa sa na overovanie kritických bodov dizajnu. Popíše sa podmienka, ktorá musí vždy platiť, a tým pádom môže odhaliť chybu. Môže pomôcť odhaliť a lokalizovať chybu vnútri dizajnu, ktorá nemusí byť patrná alebo je ťažko dohľadateľná zo samotných výstupov simulácií a verifikácií,
- **spolupráca s inými programovacími jazykmi** – SystemVerilog DPI popísané v časti 2.6.

2.3 Netcope P4

Jedná sa o jeden z produktov spoločnosti Netcope Technologies a.s.[1]pre akceleračné sieťové karty osadené FPGA čipom. Netcope P4 využíva vysoko-úrovňový jazyk P4, popísaný v kapitole 1, pomocou ktorého umožnuje popísať spracovanie paketov na FPGA platforme.

Benefitom je webové rozhranie ponúkajúce firmware ako servis (angl.: FaaS – Firmware as a Service), kedy je prechod z jazyka P4 na úroveň VHDL až po samotný firmware pre užívateľa úplne transparentný.

Pre mapovanie jazyka P4 na VHDL sa využíva kompilátor, ako je možné vidieť na obrázku 2.4. Po nahratí P4 zdrojových kódov ako úlohy do webového rozhrania začne kontrola korektnosti P4 kódu, po ktorej úspešnom absolvovaní systém spúšta nástroje na syntézu a generovanie firmwaru, ktorý si užívateľ následne môže stiahnut a nahrať do FPGA čipu [10, 11].



Obr. 2.4: Schéma P4/VHDL [12]

[1] `https://www.netcope.com/`

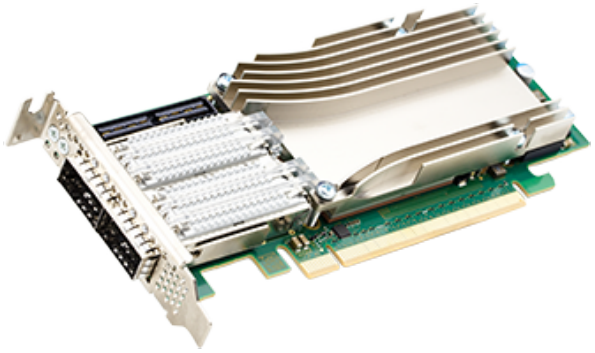
2.4 Sieťové karty s FPGA čipom

Pre vývoj akceleračných nástrojov sú v dnešnej dobe často využívané FPGA čipy. Keďže Netcope P4, popísaný v 2.3, cieli na spracovanie sieťovej prevádzky, je nevyhnutné pripojenie do siete. Preto vznikli sieťové akceleračné karty osadené FPGA čipom. To umožnuje spracovávať sieťovú prevádzku na vysokých rýchlostiach pomocou FPGA čipu. Taktiež sprostredkováva komunikáciu s hostiteľským softvérom pomocou pripojenia cez PCIExpress zbernicu.

Táto sekcia teda popisuje akceleračné sieťové karty podporované Netcope P4 produktom.

NFB-200G2QL

Platforma NFB alebo Netcope FPGA Board a jej najnovšia a najvýkonnejšia karta NFB-200G2QL. Jedná sa teda o kartu od spoločnosti Netcope Technologies a.s., ktorá podporuje rýchlosti až do 200 Gb/s. Karta je zobrazená na obrázku 2.5 a tabuľka 2.1 popisuje jej konkrétne parametre [10, 13].



Obr. 2.5: Karta NFB-200G2QL [13]

Tab. 2.1: Parametre karty NFB-200G2QL [13]

Sieťové rozhranie	2×QSFP28
PCI Express	Gen 3 x16 + x16 (128 + 128 Gb/s)
FPGA čip	Xilinx Virtex UltraScale+
Pamäťe na doske	3×72Mb QDRIIIe SRAM / 3×288Mb QDRIIIe SRAM

SILICOM FB2CGG3

Karta od spoločnosti Silicom[2] vo variante s dvomi portami a FPGA čipom typu VUP9 je podporovaná produktom Netcope P4. Obrázok 2.6 zobrazuje samotnú kartu a tabuľka 2.2 popisuje jej technickú špecifikáciu [10, 14].

Obr. 2.6: Silicom FB2CGG3[3]

Tab. 2.2: Parametre karty NFB-200G2QL [14]

Sieťové rozhranie	2X QSFP28
PCI Express	16-lane PCIe, 1-16 lane PCIe Gen1/Gen2/Gen3 via on-board PLX PCIe switch
FPGA čip	Virtex UltraScale+ VU9P
Pamäťe na doske	2 x 64-bit DDR4 2400MT/s 4GB, 2 SODIMM sockets for use with DDR4, QDRII+ or SQIVe modules

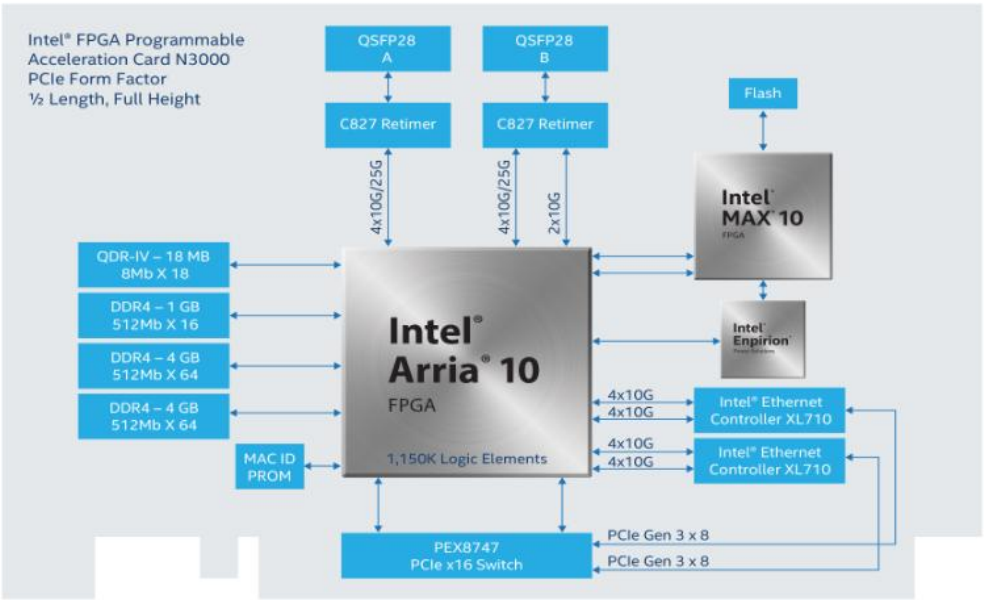
Intel® FPGA PAC N3000

Jedná sa o kartu od spoločnosti Intel. Názov PAC, alebo Programmable Acceleration Card, a teda programovateľná akceleračná karta trefne popisuje jej využitie v praxi. Je to full-duplexná 100 Gb/s podporujúca sieťová karta. Obrázky 2.7 a 2.8 znázorňujeú kartu a jej vnútornú architektúru. Tabuľka 2.3 zase jej špecifikáciu [10, 15].

[2]https://www.silicom-usa.com

[3]Obrázok dostupný z: `https://www.netcope.com/en/products/netcopep4`

Obr. 2.7: Karta Intel® FPGA PAC N3000 [15]



Obr. 2.8: Architektúra karty Intel® FPGA PAC N3000 [15]

Tab. 2.3: Parametre karty NFB-200G2QL [15]

Sieťové rozhranie	2X QSFP
PCI Express	PCI Express* (PCIe*) Gen3 x16
FPGA čip	Intel® Arria® 10 GT FPGA
Pamäťe na doske	9 GB DDR4, 144 Mb QDR-IV

2.5 FrameLinkUnaligned rozhranie

FrameLinkUnaligned alebo FLU je komunikačný protokol navrhnutý pre komunikáciu pomocou širokej datovej zbernice. Jedná sa o proprietárny protokol spoločnosti Netcope Technologies a.s. V Netcope P4 sa využíva napríklad na prenos užitočných dát paketu. Je použité usporiadanie big-endian, nakoľko je protokol cielený na paketové prenosy, a teda prvý bajt sa nachádza vpravo a nasledujúce bajty rastú smerom doľava. V tabuľke 2.4 sú popísané jednotlivé signály tohto rozhrania. Všetky signály sú synchrónne na nábežnú hranu hodín [16]. V tejto práci bude pomocou FLU rozhrania prebiehať komunikácia a prenos užitočných dát paketu do hashovacej funkcie.

Tab. 2.4: Tabuľka popisujúca signály FLU rozhrania [16]

názov signálu	**strana**	**popis**
DATA	odosielateľ	samotné dáta
SOP	odosielateľ	indikuje začiatok paketu v danom FLU slove a validitu SOP_POS signálu
SOP_POS	odosielateľ	indikuje pozíciu začiatku paketu v DATA signále, pričom je vždy násobkom ôsmich bajtov
EOP	odosielateľ	indikuje koniec paketu v danom FLU slove a validitu EOP_POS signálu
EOP_POS	odosielateľ	indikuje pozíciu konca paketu v DATA signále, pričom je vždy zarovnané na jeden bajt
SRC_RDY	odosielateľ	indikuje stav, kedy je odosielateľ pripravený na prenos dát
DST_RDY	prijímateľ	indikuje stav, kedy je prijímateľ pripravený na prenos dát

2.6 SystemVerilog Direct Programming Interface

Skratka DPI alebo Direct Programming Interface označuje rozhranie medzi jazykom SystemVerilog, popísaným v 2.2, a vzdialeným, vyšším programovacím jazykom. Pozostáva z dvoch izolovaných strán a to stranou SystemVerilogu a stranou iného programovacieho jazyka. Momentálne je natívna podpora iba pre jazyk C.

Využíva sa princíp black box, kedy špecifikácia a vnútorná implementácia komponenty sú oddelené. Implementácia je teda pre systém transparentná a tým pádom aj jej jazyk [8].

Konkrétne použitie a postup ako na to je popísaný v jednej z následných časti, konkrétne v sekcii 4.2.

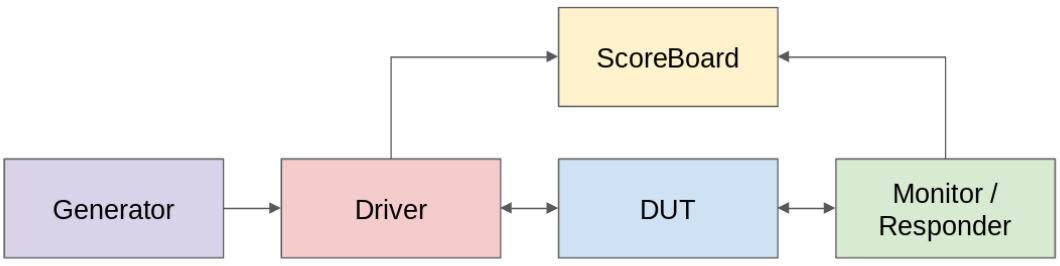
2.7 Simulácia a verifikácia

Vzhľadom na výskyt chýb objavujúcich sa pri návrhu zložitých obvodov a náročnosť ich odhalenia a opravenia sa verifikácie a simulácie stávajú neoddeliteľnou súčasťou tohoto procesu. Jedná sa o kroky pomáhajúce odhaliť chyby v návrhu.

`Simulácia` je jedným z procesov na overenie správnej funkcionality návrhu. Na vstupné rozhranie prichádza konečná množina dopredu definovaných vstupných dát, ktoré sa postupne predávajú na vstup testovanej komponente. Najčastejšie sú definované priamo v zdrojovom súbore simulácie (angl.: testbench). Najčastejšie je výstupom simulácie vizuálna kontrola jednotlivých hodnôt výstupných signálov a medzivýsledkov. Týmto postupom je možné odhaliť chyby návrhu, avšak jedná sa len o overenie konkrétnych vstupov v konkrétnom čase, a tým pádom možnosti odhalenia limitujúceho počtu chýb.

`Funkčná verifikácia` je proces vychádzajúci so simulácií používajúci zložitejšie, viac sofistikovanejšie mechanizmy na odhalenie chýb v návrhu. Jedná sa napríklad o využitie náhodného generovania vstupných dát, sekcia 2.2 a obmedzené náhodné generovanie stimulov, alebo mechanizmu na automatické overenie výpočtov a správnosti vstupov [9, 17].

Na obrázku 2.9 je zobrazená architektúra verifikačného prostredia CESNET, ktoré bude následne v práci využité a na základe ktorého princípov bude tvorená verifikácia.



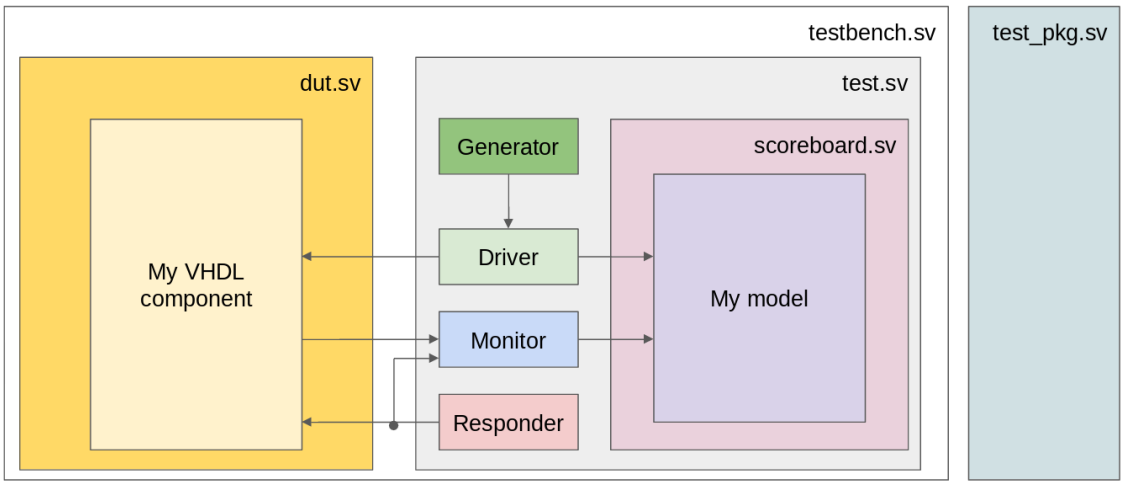
Obr. 2.9: Diagram verifikácie [18]

Jednotlivé bloky diagramu 2.9 môžeme popísať ako [9]:

- **Generator** – jednotka produkujúca obmedzené, náhodne generované stimuly a to vstupné dáta a oneskorenia (angl.: delay) v a medzi transakciami.

- **Driver** – prijaté transakcie od jednotky Generator prevádzajú na konkrétne rozhranie a vkladajú ako vstupné dáta testovaného návrhu. Výstup Driver jednotky teda putuje ako vstup do testovaného návrhu a zároveň do jednotky ScoreBoard.
- **DUT** (Design Under Test, sk: testovaný návrh) – testovaná komponenta návrhu, ktorá v softvérovej verifikácii beží v prostredí simulátoru. Je pripojená pomocou vstupného a výstupného rozhrania k verifikačnému prostrediu.
- **Monitor** – prevádza výstupy DUT jednotky konkrétneho rozhrania na transakcie, ktoré následne putujú do ScoreBoard jednotky.
- **Responder** – jednotka starajúca sa o odpovedanie na výstupnom rozhraní, pomocou ktorej je možné vkladať definované oneskorenia.
- **ScoreBoard** – obsahuje model testovaného návrhu, v abstraktnejšom jazyku SystemVerilog alebo využíva implementáciu v jazyku C pomocou SystemVerilog DPI. Do modelu vstupujú dáta z Driver jednotky, ktoré sa modelom spracujú rovnako, ako by sa mali spracovať v testovanom návrhu. Následne sa výstup tohoto modelu porovnáva s výstupom DUT jednotky, a tak sa verifikuje správnu funkcionalitu testovaného návrhu.

Obrázok 2.10 ďalej popisuje hierarchickú štruktúru súborov verifikačného prostredia CESNET a zapojenie testovaného návrhu pomenovaného ako `My VHDL component`.



Obr. 2.10: Štruktúra súborov verifkácie [18]

Toto verifikačné prostredie pracuje na princípe výpočtu v testovanej komponente a následne v modeli komponenty nachádzajúcom sa v bloku ScoreBoard. Model po výpočte danú transakciu uloží do tabuľky. Pri prijatí transakcie z testovanej komponenty sa tabuľka prehľadá, a ak sa v nej nachádza rovnaká transakcia, čo znamená že sa oba výstupy rovnajú, odstráni sa z danej tabuľky. Po dokončení

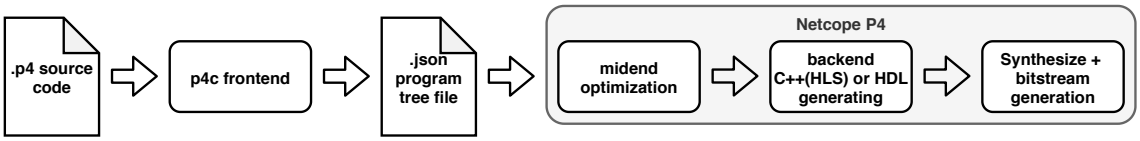
testovacieho scenára sa tabuľka vypíše, a ak verifikácia prebehla v poriadku, a teda výpočet v testovanej komponente sa pre každú vstupnú transakciu rovnal výpočtu z modelu, je tabuľka prázdna. Ak sa transakcia v danej tabuľke nenájde, a teda výpočet testovanej komponenty je zlý, verifikácia sa ukončí s chybou. Ak po ukončení testovacieho scenára zostane v tabuľke nejaká transakcia, znamená to, že sa určité transakcie neprijali, respektíve ich testovaná komponenta neposlala na výstup, a teda nastala chyba a strata transakcie počas výpočtu.

3 Mapovanie externých objektov do P4 zreťazenia

Kapitola na začiatku predstavuje kompilátor jayzka P4 do VHDL, architektúru samotného P4 zreťazenia (angl.: pipeline) a následne popisuje rozbor a možné spôsoby implementácie (mapovania) externých objektov do nej.

3.1 Kompilátor $P4_{16}$–VHDL

Kompilátor jayzka $P4_{16}$ do jayzka VHDL, respektíve mapovanie na FPGA platformu je výsledkom spolupráce spoločnosti Netcope Technologies a.s. a združenia CESNET[1]. Je vytvorený na základe referenčného kompilátoru p4c [20], a napísaný v jazyku C++. Využíva modulárnu architektúru, ktorá sa skladá z troch vrstiev a to frontend, midend a backend. Midend vrstva prešla úpravou a stará sa o optimalizácie. Následne bol vytvorený platformne závislý backend, ktorý dovoľuje kompilovať jazyk P4 do VHDL [20]. Schéma využitia kompilátoru je zobrazená na obrázku 3.1.

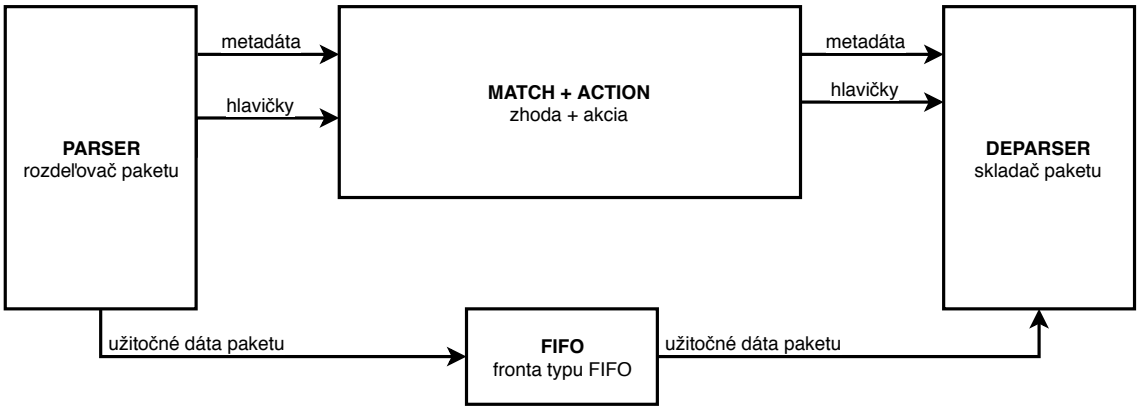


Obr. 3.1: Proces kompilácie

3.2 P4 zreťazenie

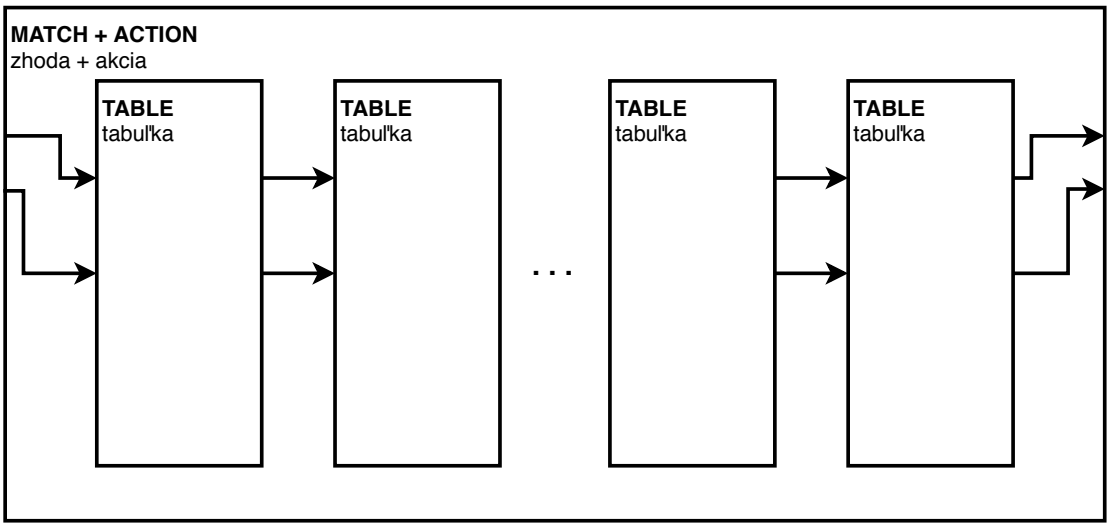
P4 zreťazenie sa skladá z troch hlavných častí. Prvou je konfigurovateľný `PARSER` (rozdeľovač paketu). V tomto bloku dochádza k rozdeleniu (angl.: parsing) paketu a extrakcii hlavičiek definovaných P4 programom do jednotlivých políčok. Následne oddeľuje užitočné dáta paketu (angl.: payload), ktoré putujú do samostatnej fronty typu FIFO (First-In First-Out), blok `FIFO`, mimo spracovanie. Oddelené hlavičky putujú do bloku zhoda + akcia, `MATCH + ACTION` blok, kde sú najskôr vyhodnocované pravidlá pre jednotlivé pakety a následne uplatňované definované akcie nad nimi. Oddelené užitočné dáta paketu na konci vstupujú spolu s upravenými hlavičkami a metadátami do konfigurovateľného `DEPARSER` bloku, časť tri, ktorý celý paket naspäť poskladá. Tento popis je znázornený na obrázku 3.2.

[1] `https://www.cesnet.cz`

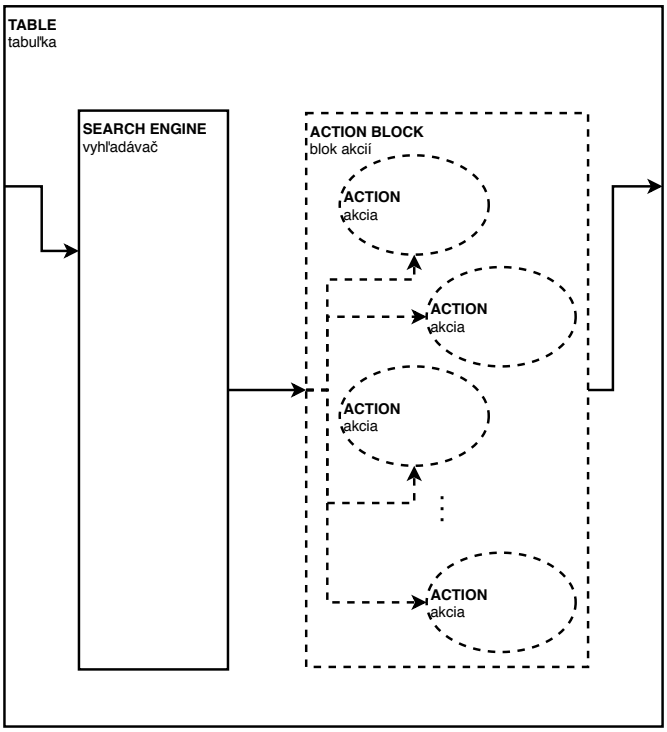


Obr. 3.2: Všeobecná architektúra P4 zreťazenia

Časť zhody + akcie, ako je možné vidieť na obrázku 3.3, sa ďalej skladá z jednotlivých tabuliek, `TABLE`, ktoré postupne za sebou spracovávajú vstupné dáta. Každá pozostáva z vyhľadávača (`SEARCH ENGINE`) a bloku akcií, `ACTION` bloku, obsahujúcom jednotlivé akcie. Vyhľadávač vyhľadáva zhodu paketu a pravidla, ktorá ak nastane, zvolí požadovanú akciu, ktorú treba vykonať a deleguje to bloku akcií. Architektúru samotnej tabuľky je možné vidieť na obrázku 3.4.



Obr. 3.3: Architektúra bloku zhoda + akcia



Obr. 3.4: Architektúra tabuliek

3.3 Mapovanie externých objektov

Keďže podpora externých objektov prichádza až v novej revízii jazyka, $P4_{16}$, ich mapovanie zatiaľ nie je vyriešené. Samotná špecifikácia definuje dva typy externých objektov, ktoré nepracujú s užitočnými dátami paketu, a to zdieľané a nezdieľané. Pre naše použitie, a teda podporu kryptografických primitív, je však nutné spracovávať aj užitočné dáta paketu a teda definujeme ďalšiu kategóriu externých objektov, ktoré s ním pracujú.

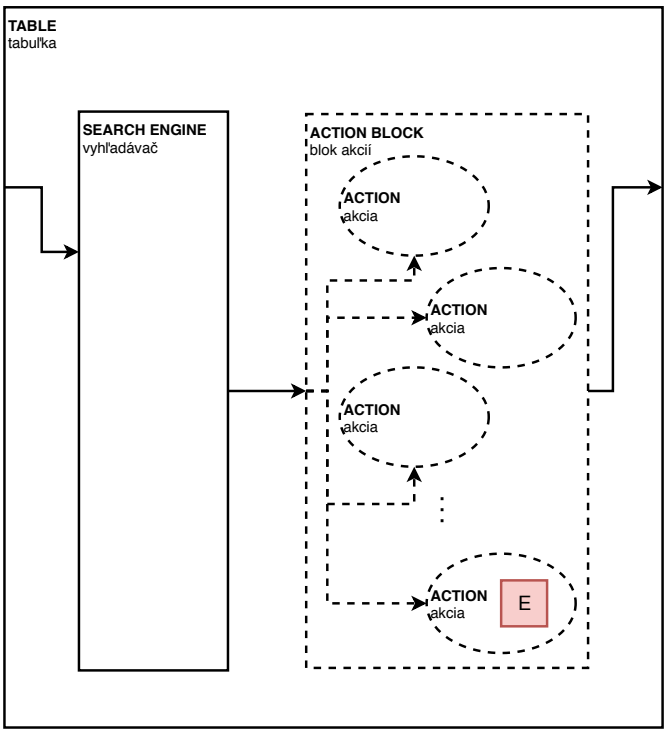
3.3.1 Všeobecné externé objekty

Externé objekty pracujúce s dátami, ktoré sú dostupné v bloku zhoda + akcia a teda nepracujú s užitočnými dátami paketu.

Nezdieľané

Jedná sa o externé objekty, ktorých funkcie sú volané iba jednou akciou. Vloženie externého objektu je znázornené na obrázku 3.5, kde blok označený ako **E** je externý objekt. Tento externý objekt je instancovaný priamo v instancii danej akcie a nie je prístupný iným akciám a tabuľkám. Je vložený pomocou vysoko-úrovňovej syntézy

(angl.: HLS – High-Level Synthesis[2]), a napísaný priamo v jazyku C alebo ako IP (Intellectual property) Core v jazyku VHDL.



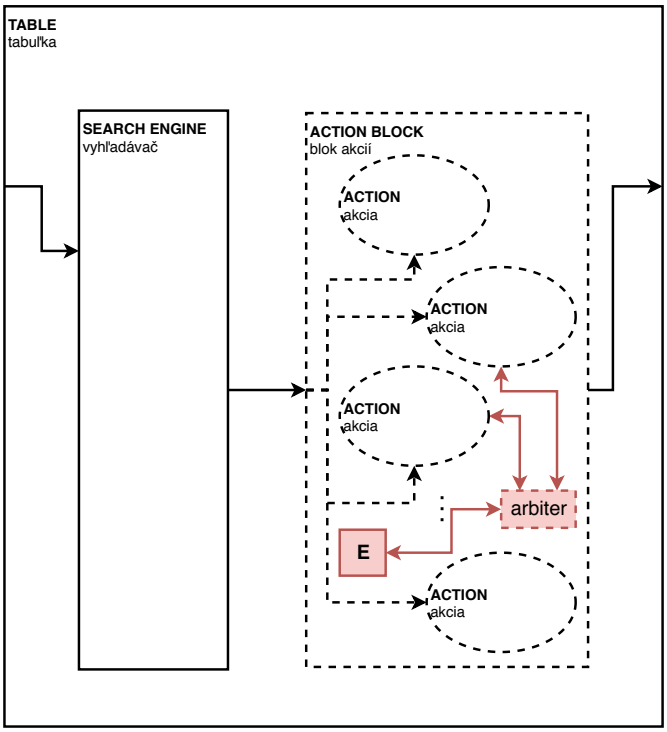
Obr. 3.5: Umiestnenie nezdieľaného externého objektu (E)

Zdieľané

U zdieľaných externých objektov je nutné riešiť zdieľaný prístup k danému externému objektu pomocou arbitra. Boli identifikované dva prípady: ak je externý objekt využívaný viacerými akciami rovnakej tabuľky a ak je využívaný viacerými akciami z rôznych tabuliek.

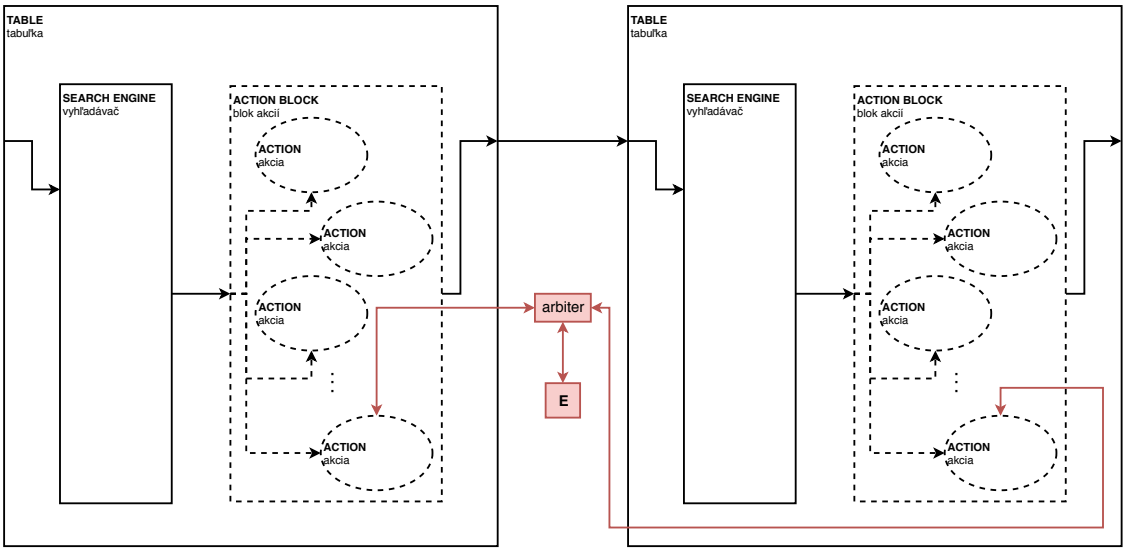
Prístup k externému objektu z rôznych akcií v rovnakej tabuľke bude riešený tak, ako zobrazuje obrázok 3.6. Keďže celý blok akcií je generovaný pomocou vysokoúrovňovej syntézy, je možné použiť ho na vyriešenie zdieľaného prístupu k externému objektu alebo navrhnúť arbiter v jazyku C, prípadne použiť jeho IP Core v jazyku VHDL.

[2]nástroj na syntézu návrhu v jazyku C do RTL dizajnu. Podpora Intel: `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. Podpora Xilinx: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`

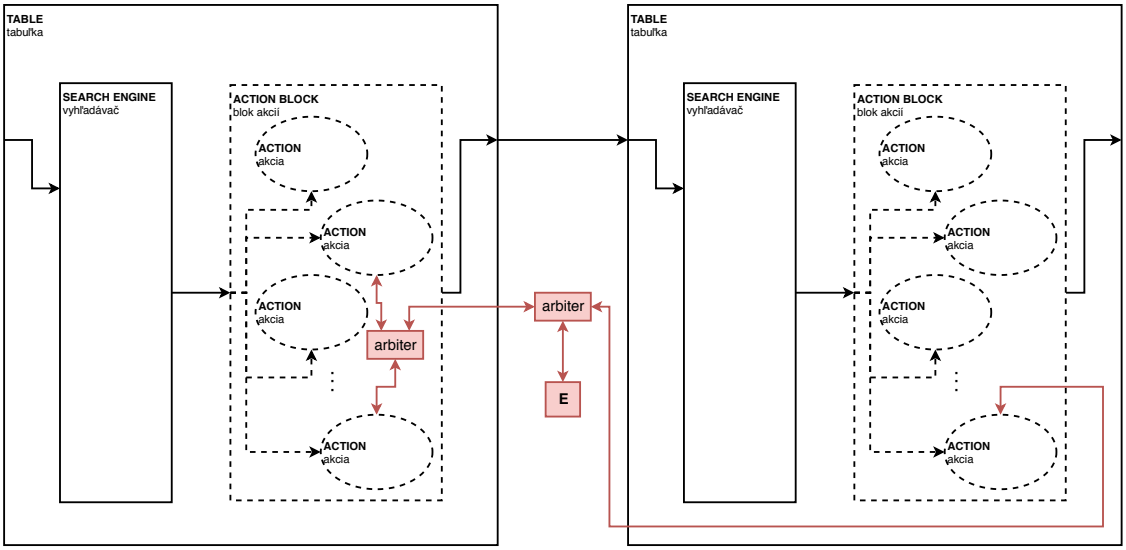


Obr. 3.6: Umiestnenie zdieľaného externého objektu (E) viacerými akciami rovnakej tabuľky

Ak je pristupované k externému objektu z viacerých akcií v rôznych tabuľkách, je nutné vyriešiť vyvedenie rozhrania zo samotnej akcie cez blok akcií a tabuľku k samotnému externému objektu, respektíve k jeho arbitru pre každú akciu, ktorá externý objekt využíva. V prípade, že externý objekt využíva viacero akcií z rovnakej tabuľky a zároveň akcie inej tabuľky, nie je nutné vyvedenie rozhrania pre každú akciu rovnakej tabuľky zvlášť. Je možné využiť interný arbiter v danej tabuľke a následne vyviesť jeho výstup k arbitru externého objektu. Túto situáciu je možné vidieť na obrázku 3.7 a 3.8.



Obr. 3.7: Umiestnenie zdieľaného externého objektu (E) využívaného viacerými akciami rôznych tabuliek

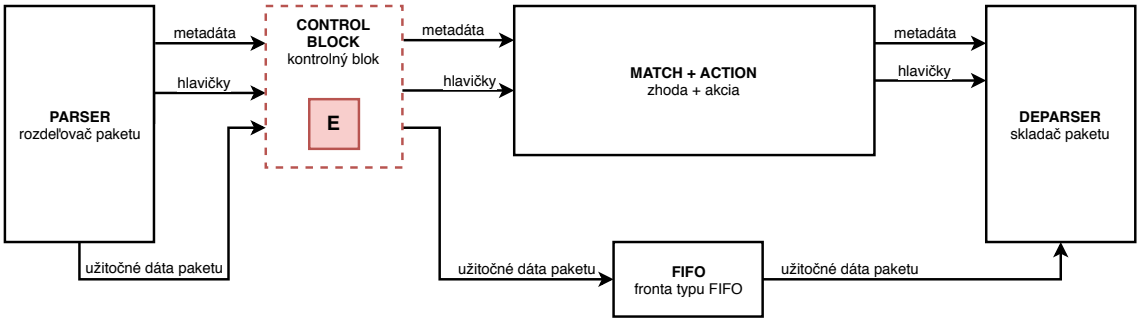


Obr. 3.8: Umiestnenie zdieľaného externého objektu (E) využívaného viacerými akciami rôznych tabuliek 2

3.3.2 Externé objekty pracujúce s užitočnými dátami paketu

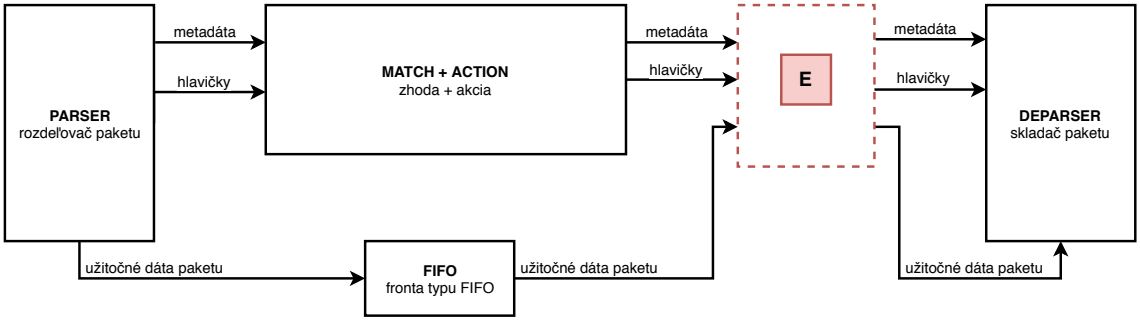
Pre vloženie externých blokov pracujúcich s užitočnými dátami paketu, čo je u kryptografických externých objektov nutnosťou, boli identifikované dva scenáre. V prvom je externý objekt vložený do konštrukcie jazyka P4 zvaného `control block`. Pomocou metódy `apply` je možné povoliť vykonávanie práce externého objektu v control

bloku. Ako konkrétne využitie môže byť použitý externý objekt na výpočet kontrolného súčtu, ktorý taktiež pracuje s užitočnými dátami paketu, a teda overenia správnosti prijatého paketu pred vstupom do časti zhoda + akcia. Tento scenár je zobrazený na obrázku 3.9.



Obr. 3.9: Umiestnenie externého objektu (E) do kontrolného bloku pred blok zhoda + akcia

Druhým scenárom je umiestnenie externého objektu za blok zhoda + akcia, kde má tým pádom k dispozícii taktiež všetky medatáta a hlavičky spolu s užitočnými dátami paketu. Tento scenár zobrazuje obrázok 3.10.



Obr. 3.10: Umiestnenie externého objektu (E) za blok zhoda + akcia

4 P4 zreťazenie s externým objektom

V tejto kapitole je popísaný výber externého objektu, ktorý je následne použitý v P4 zreťazení ako „proof of concept“[1] návrhu podpory kryptografických externých objektov a ich umiestnenia v P4 zreťazení. Popisuje jeho nutné úpravy, jednotlivé simulácie a obálku pre štandardné FLU rozhranie. Ďalej popisuje samotné P4 zreťazenie a využitie externého objektu v ňom. Následne popisuje vytvorenie a verifikáciu tohto zreťazenia s externým objektom.

4.1 Externý objekt

Ako externý objekt bola zvolená hashovacia funkcia, ktorá bude následne v P4 zreťazení počítať hash z užitočných dát paketu. Po prvotnej rešerši zdrojov a doporučení bola zvolená konkrétna hashovacia funkcia `SHA3-512`.

Jedná sa o hashovaciu funkciu doporučovanú organizáciou NIST (National Institute of Standards and Technology[2]) ako štandard bezpečnosti. [19]

4.1.1 IP Core

Ako IP Core bola zvolená implementácia z opencores[3], čo je komunita pre otvorené a Open Source IP Cores[4]. Z implementácií kryptografických jadier (angl.: crypto core) bola zvolená `SHA3 (KECCAK)`[5]. Jedná sa o OpenCores certifikovaný projekt. Obsahuje dokumentáciu a dve implementácie. Bola zvolená `high_throughput_core`, ktorá je optimalizovaná pre vyššiu priepustnosť a frekvencie. IP Core je distribuovaný pod licenciou Apache License, verzia 2[6], a teda copyrightom: Copyright 2013, Homer Hsing <homer.hsing@gmail.com>. Zdrojové súbory sú napísané v jazyku verilog.

Úprava implementácie

Pri testovaní implementácie pomocou testovacích vektorov[7] a štúdii špecifikácie hashovacej funkcie SHA3–512 bolo zistené, že implementácia nekorektne počíta hash. Následne bolo zistené, že hash je korektný ako výstup hashovacej funkcie

[1] SK: dôkaz koncepcie

[2] `https://www.nist.gov/`

[3] `https://opencores.org/`

[4] IP Core s otvoreným zdrojovým kódom pod voľnou licenciou

[5] `https://opencores.org/projects/sha3`

[6] `https://www.apache.org/licenses/LICENSE-2.0`

[7] `https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Secure-Hashing#sha3vsha3vss`

keccak-512. SHA funkcie sú definované z funkcie keccak[c]. Prevod medzi funkciou SHA3–512 a keccak–512 je v pridaní bitov 01 za správu M [19]:

$$SHA3-512(M) = KECCAK[1024](M||01, 512) \tag{4.1}$$

Po podrobnejšom zoznámení sa so zdrojovými kódmi a funkcionalitou jednotlivých blokov a častí, bolo možné prejsť k náprave. Došlo teda k úprave implementácie daného IP Coru a to konkrétne zdrojového súboru padder1.v a teda modulu padder1, príloha B, ktorý pridáva výplň (angl.: padding) a je v ňom možné pridať vyššie spomínané dva bity 01 za správu M a teda vytvoriť z funkcie keccak–512 funkciu SHA3–512. K úprave došlo aj pridaním generického parametra IS_SHA, v module keccak, padder a padder1, pomocou ktorého je možné meniť funkcionalitu. Ak je IS_SHA = 0 implementácia počíta hash keccak–512, ak je IS_SHA = 1, čo je aj východzia hodnota tohto parametru, počíta hash SHA3–512.

Táto úprava bola reportovaná ako chyba (angl.: bug) k samotnému IP Coru ako Hash not work correctly as SHA3-512 #3[8].

4.1.2 Referenčná implementácia v jazyku C

Ako referenčná implementácia v jazyku C bola zvolená The eXtended Keccak Code Package, v skratke XKCP. Ide o jednu z doporučovaných implementácii tímom Keccak[9]. Pozostáva zo súboru rôznych nezávislých implementácií založených na Keccak a Xoodoo schémach ako napríklad cSHAKE, KMAC alebo SHA3. Jedná sa o otvorené implementácie dostupné na platforme GitHub[10]. Pre náš účel bola vybraná konkrétne nezávislá (angl.: standalone) implementácia Keccak-more-compact.c. Je distribuovaná ako verejná doména (angl.: public domain) asociovaná listine Creative Comons Public Domain[11].

Bude používaná konkrétne funckia FIPS202_SHA3_512. Rozhranie funkcie je definované ako:

```
void FIPS202_SHA3_512(const u8 *in, u64 inLen, u8 *out) {
    Keccak(576, 1024, in, inLen, 0x06, out, 64);
}
```

, kde sú jednotlivé typy definované ako:

[8] https://opencores.org/projects/sha3/issues/3

[9] https://keccak.team/software.html

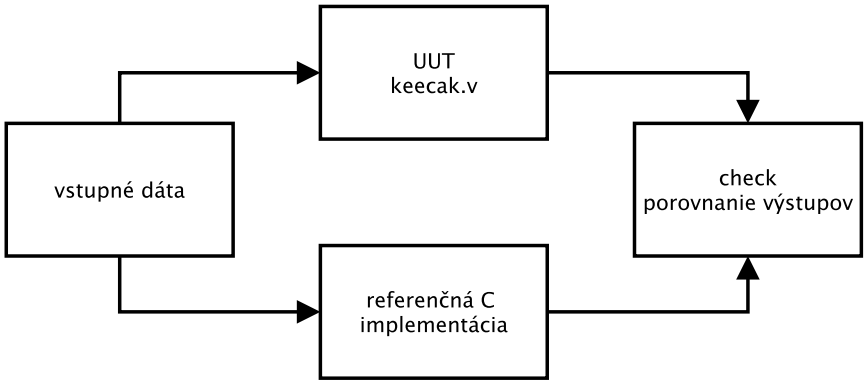
[10] repozitár dostupný na: https://github.com/XKCP/XKCP

[11] https://creativecommons.org/publicdomain/zero/1.0/

```
typedef unsigned char u8;
typedef unsigned long long int u64;
```

4.2 Simulácia IP Coru

Simulácia IP Coru prebiehala, ako je znázornené na obrázku 4.1, overením voči referenčnej implementácii v jazyku C popísanej v predchádzajúcej časti. Ide o overenie pomocou niekoľkých ručne definovaných vstupných dát, ktoré vstupujú zároveň do simulovaného modulu keccak.v, označeného ako UUT[12] a referenčnej funkcie v jazyku C pomocou SystemVerilog DPI. Ich výstupy sa následne porovnajú, a tým overia správnu funkcionalitu testovaného modulu. Výstup tejto simulácie je možné vidieť na výpise v prílohe D.



Obr. 4.1: Schéma simulácie

Na to bolo potrebné upraviť samotný testbench pre IP Core, rozhranie funkcie v jazyku C a použiť SystemVerilog DPI, popísaný v sekcii 2.6.

Vzhľadom na použitie a kompatibilitu rozhrania medzi SystemVerilogom a jazykom C bolo nutné upraviť rozhranie funkcie v jazyku C respektíve jej datové typy a to tak, ako je zobrazené vo výpise v prílohe C. Toto rozhranie je možné porovnať s pôvodným, zobrazeným vo výpise 4.1.2. Ďalej bola do funkcie pridaná možnosť definovať makro `DEBUG`, a tým pádom využiť pridané pomocné výpisy a sledovať prácu tejto funkcie, jej vstupy a výstupy. Tieto zmeny je taktiež možne vidieť v spomenutej prílohe.

Ďalším krokom bola úprava samotného testbenchu. Ako prvé prišiel na rad import funkcie jazyka C ako:

[12]Unit Under Test – testovaná časť

```
import "DPI-C" function void FIPS202_SHA3_512(
    input string in_value,
    input longint unsigned in_len,
    output byte unsigned out_value[64]
    );
```

, kde bolo potrebné správne nadefinovať dátové typy tak, aby korešpondovali s dátovými typmi vo funkcii v jazyku C. Tým pádom bolo vytvorené kompatibilné rozhranie tak, ako je znázornené v tabuľke 4.1.

Tab. 4.1: Kompatibilita rozhrania medzi SystemVerilogom a C jazykom pre DPI

SystemVerilog	**C**
input string	char *
input longint unsigned	uint64_t
output byte unsigned <var_name> [64]	unsigned char * <var_name> [64]

Pre overenie kompatibility, rozhranie a prípadné ladenie a hľadanie rozdielov je možné použiť vygenerovanie `dpiheader` hlavičky z testbenchu v SystemVerilogu a následne porovnať s rozhraním funkcie v jazyku C. Vygenerovaný hlavičkový súbor dpiheader.h popisuje rozhranie funkcie tak, ako je reprezentované v systemverilogu a preklopené do jazyka C. Laicky povedané tak, ako ho chápe SystemVerilog pri kompilácii. Je možné tak urobiť pomocou:

```
$ vlog -dpiheader <path>/dpiheader.h <testbench_file>.sv
```

Následne je nutné skompilovať zdrojové súbory funkcií v jazyku C pred samotnou simuláciou. Bol zvolený spôsob kompilácie priamo v simulačnom prostredí ModelSim.

```
$ vlog -sv <testbench_file>.sv <C_file_name>.c
```

Všetky vyššie popísané kroky boli automatizované pomocou vytvorenia `.do` súboru, ktorý je následne možné spustiť jedným príkazom ModelSimu, automaticky vytvoriť dpiheader.h hlavičkový súbor, skompilovať zdrojové C súbory funkcií a následne spustiť simuláciu príkazom:

```
$ vsim -c -do <do_file_name>.do
```

, kde parameter -c nastavuje použitie čisto konzolového výstupu. Samotný do súbor je zobrazený na výpise 4.1.

Výpis 4.1: Do súbot pre simuláciu v konzoli

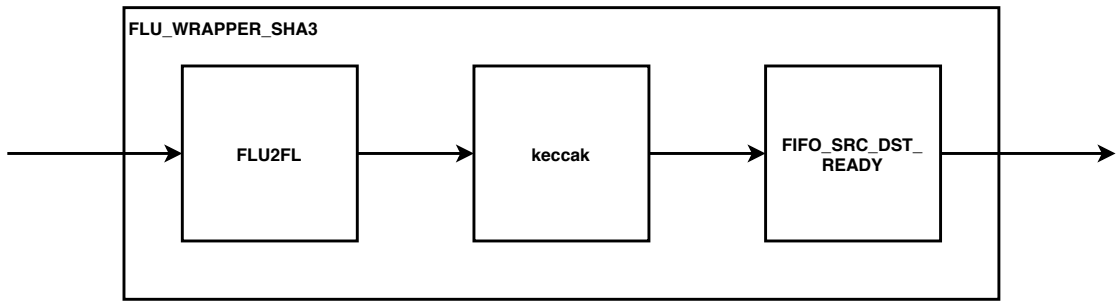
```
1 vlib work
2 vlog -lint ../rtl/*.v
3 vlog -dpiheader <C_path>dpiheader.h test_keccak.sv
4 vlog -sv test_keccak.sv <C_path>/Keccak-more-compact.c
5 vsim -novopt test_keccak
6 run -all
```

V prípade potreby je možné pridávať signály zobrazené v bloku, kde sú signály zobrazené ako vlny (angl.: waveform) a tiež pomocou .do súboru a to takto:

```
add wave -noupdate -format <data_format> -radix /
<radix_type> <unit_name>/<signal_name>
```

4.3 FrameLinkUnaligned obálka na IP Core

Obálka s FLU vstupným rozhraním bola vytváraná z dôvodu nutnosti práce s užitočnými dátami paketu. Jej schéma je zobrazené na obrázku 4.2. Ako prvé vstupné FLU dáta prechádzajú cez komponentu `FLU2FL`[13], ktorá ich zarovná. Následne sú postupne nasekané a vstupujú po jednotlivých blokoch do hashovacej komponenty `keccak`.

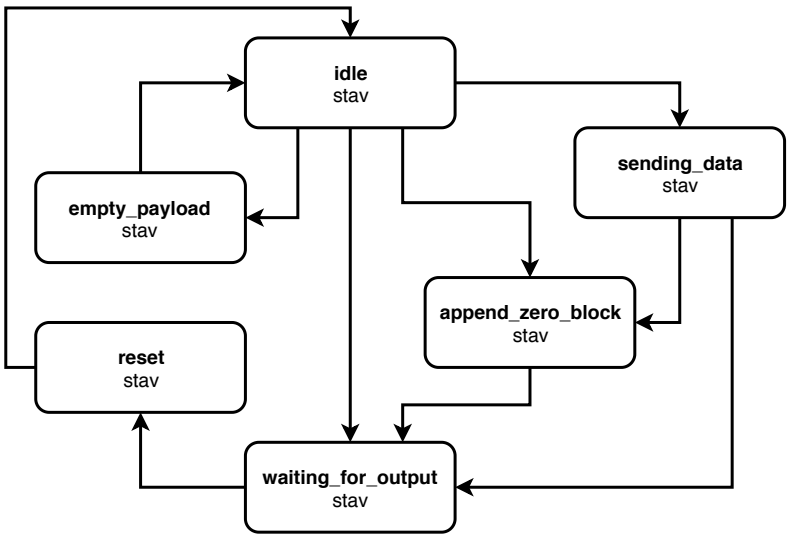


Obr. 4.2: Abstraktná schéma architektúry FLU obálky IP Coru

O synchronizáciu blokov a správne nastavenie signálov pre komponentu keccak sa stará konečný stavový automat (angl.: FSM – Finite State Machine). Jeho stavy

[13]komponenta prevádzajúca nezarovnané FLU na zarovnané FL – začiatok paketu sa môže nachádzať vždy iba na začiatku nového FL slova. Dostupná z repozitára fwbase ©CESNET

a jednotlivé prechody sú znázornené na obrázku 4.3. Stav `idle` je stavom kedy sa očakáva valídny vstup. Ak táto situácia nastane, do komponenty keccak vstupuje prvý blok dát. Taktiež sa na základe vstupu zvolí nasledujúci stav do ktorého automat vstúpi. `empty_payload` je stavom pre situáciu, kedy je potrebné hashovat prázdne užitočné dáta paketu a teda hash je pre daný paket nevalídny. Stav `sending_data` zabezpčuje posielanie jednotlivých datových blokov slova do komponenty keccak a signalizuje tak prebiehajúce spracovanie daného slova. Vzhľadom na architektúru komponenty keccak v prípade dĺžky paketu rovnej násobku šesťdesiatštyri bitov potrebné na konci paketu vložiť ešte jeden konečný, prázdny blok. Na túto situáciu je tu stav `append_zero_block`. Po odoslaní posledného bloku je potrebné čakať na výstup hashovania. To sa deje v stave `waiting_for_output`. Po signalizácii výstupu je nutné komponentu resetovať a tým pádom pripraviť na další, nový paket stavom `reset`.



Obr. 4.3: Stavy a prechody konečného stavového automatu FLU obálky IP Coru

Výstupný hash z komponenty keccak vstupuje do komponenty `FIFO_SRC_DST_READY`. Jedná sa o komponentu vytvorenú za účelom zaobalenia obyčajnej fronty typu FIFO na prácu so synchronizačnými signálmi (angl.: handshake). Je postavená nad komponentami/využíva komponenty `FIFO`[14]prípadne `FIFO_BRAM`[14] podľa nastavenia typu pamäte. Z tejto fronty napokon výstup putuje ako výstup samotnej komponenty obálky `FLU_WRAPPER_SHA3`.

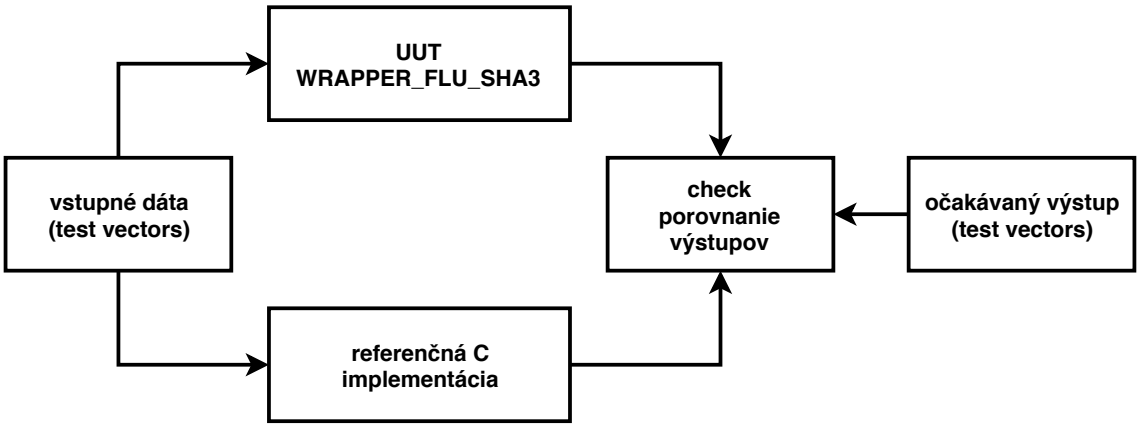
Výpis rozhrania FLU obálky IP Coru je možné vidieť v prílohe E. Má základné kontrolné vstupy ako hodiny, `CLK`, a reset, `RESET`. Ďalej definované vstupné FLU rozhranie pomocou signálov `RX_DATA`, `RX_SOP`, `RX_SOP_POS`, `RX_EOP`, `RX_EOP_POS`,

[14]Dostupná z repozitára fwbase ©CESNET

RX_SRC_RDY a RX_DST_RDY. Význam jednotlivých FLU signálov je popísaný v časti 2.5. Výstupné rozhranie tvoria signály slúžiace na prenos výstupného hashu. HASH_DATA prenáša samotné užitočné dáta, respektíve hodnotu hashu. Ďalej sú tu synchronizačné signály HASH_VLD, značiaci pripravenosť strany odosielateľa, a HASH_DST_RDY, značiaci pripravenosť strany prijímateľa. Posledným signálom je HASH_OK identifikujúci správnosť počítaného hashu. Ak je paket bez užitočných dát, respektíve užitočné dáta neobsahuje, hash sa nepočíta a označí sa HASH_OK = 0.

4.3.1 Simulácia

Architektúra simulácie je zobrazená na obrázku 4.4. Ako vstupy sú použité testovacie vektory zobrazené v tabuľke 4.2, ktoré vstupujú do testovanej komponenty ako aj do referenčnej implementácie v jazyku C. V check časti sa následne výstup oboch implementácií overí voči očakávanému výstupu testovacích vektorov. Výstup simulácie je zobrazený vo výpise v prílohe F.



Obr. 4.4: Schéma simulácie

Tab. 4.2: Testovacie vektory[15]simulácie

vstup	**dĺžka vstupu**
h'6e0c65ee0943e34d9bbd27a8547690f2291f5a86d713c2be258e6ac169/ 19fe9c4d491895d3a961bb97f5fac255891a0eaa18f80e1fa1ebcb639fcfc1	480b
h'bcc9849da4091d0edfe908e7c3386b0cadadb2859829c9dfee3d8ecf9dec8/ 6196eb2ceb093c5551f7e9a4927faabcfaa7478f7c899cbef4727417738fc06	496b
h'56ac4f6845a451dac3e8886f97f7024b64b1b1e9c5181c059b5755b9a6042be/ 653a2a0d5d56a9e1e774be5c9312f48b4798019345beae2ffcc63554a3c69862e	512b

4.3.2 Syntéza

Pre syntézu bol zvolený FPGA čip karty NFB-200G2QL popísanej v časti 2.1, a to konkrétne Xilinx Virtex UltraScale+ – xcvu7p-flvb2104-2-i. Cieľová frekvencia bola nastavená na 200 MHz a teda časovú periódu hodín 5 ns. Výsledky časovej analýzy sú zobrazené na obrázku 4.5, kde kladné WNS znamená že obvod splňuje časovanie, a výsledky analýzy zabraných zdrojov v tabuľke 4.3.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2.423 ns	Worst Hold Slack (WHS):	0.059 ns	Worst Pulse Width Slack (WPWS):	1.968 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	14033	Total Number of Endpoints:	14033	Total Number of Endpoints:	4436

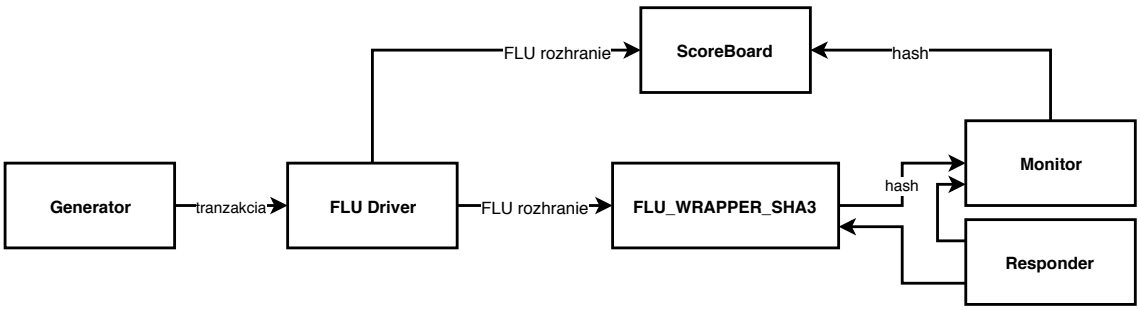
Obr. 4.5: Výsledky analýzy časovania

Tab. 4.3: Výsledky analýzy zabraných zdrojov

Typ zdroja	**Použité zdroje**	**Dostupné zdroje**	**Použité zdroje [%]**
CLB LUTs	8906	788160	0.0113
CLB Registers	3844	1576320	0.0024
CARRY8	11	98520	0.0001

4.3.3 Verifikácia

Verifikácia overuje správnosť výpočtu hashu IP coru s FLU rozhraním, a teda samotnú komponentu FLU_WRAPPER_SHA3. Schéma zapojenia verifikácie je na obrázku 4.6. Ako je možné vidieť, vstupy sú obmedzene náhodne generované. Nastavenie samotnej verifikácie je zobrazené a popísané nižšie.



Obr. 4.6: Abstraktná schéma blokov verifikácie FLU_WRAPPER_SHA3

[15] https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Secure-Hashing#sha3vsha3vss

Jednotlivé vstupy sú generované ako paketové transakcie v bloku `Generator`. Následne sú prevádzané do FLU transakcií v bloku `FLU Driver`, z ktorého putujú na vstup testovanej komponenty, blok `FLU_WRAPPER_SHA3` a do bloku `ScoreBoard`. Testovaná komponenta zo vstupnej FLU transakcie vypočíta hash, ktorý ako výstup putuje do bloku `Monitor`. `Responder` je blok starajúci sa o chod výstupných transakcií. Z Monitor bloku následne putuje hash do bloku ScoreBoard. V tomto bloku prebieha výpočet rovnakého hashu zo vstupnej FLU transakcie za pomoci hashovacej funkcie napísanej v jazyku C, ktorá slúži ako referenčná (angl.: golden model). Oba výstupy sú následne porovnané, čím dôjde ku kontrole správnosti hashu a teda jeho výpočtu v komponente.

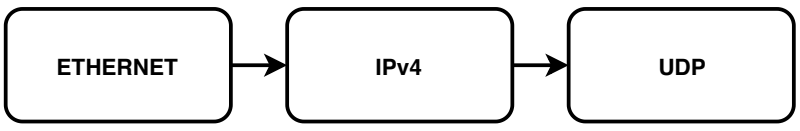
Na testovanie je použitých päť verifikačných scenárov zobrazených v prílohe G vo výpise G.1. Každému scenáru sú nastavené určité parametre, ako napríklad najmenšia a najvačšia dĺžka paketu, širka vstupnej FLU zbernice a obmedzenia generovania signálov blokom Driver a Responder. Následne je nastavený počet transakcií, ktorý je pre každý scenár 20 000.

Celá verifikácia prebehla úspešne pre všetkých 100 000 transakcií a jej výstup je zobrazený v prílohe G vo výpise G.2.

Táto verifikácia teda overuje správnosť výpočtu samoného hashu SHA3-512 zo vstupných dát prichádzajúcich cez FLU rozhranie, a teda aj samotnú funkcionalitu obálky na IP Core.

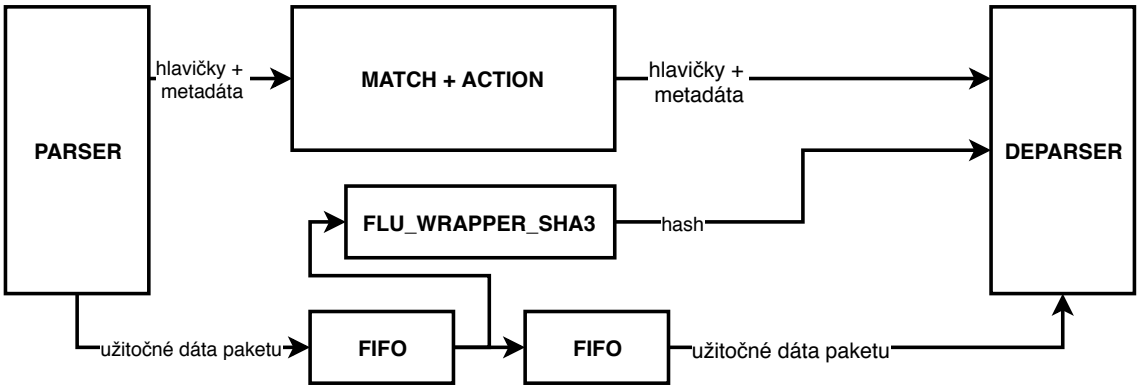
4.4 Verifikácia P4 zreťazenia s externým objektom

Pre vloženie externého objektu bol zvolený scenár popísaný v sekcii 3.3.2, konkrétne vloženie externého objektu za blok zhoda + akcia, zobrazený na obrázku 3.10. Externý objekt je vložený do jednoduchého P4 zreťazenia, ktoré definuje jednoduchý parser zobrazený na obrázku 4.7, a teda popisujúci hlavičky protokolu ethernet, IP verzie 4 a UDP protokolu. Obsahuje jednu tabuľku a jednu akciu, ktorá sa stará o výpočet kontrolného súčtu UDP paketu, avšak nie je uplatňovaná. Tabuľka je iba povolená v základnej operácii a tým pádom nijako nezasahuje do paketov.



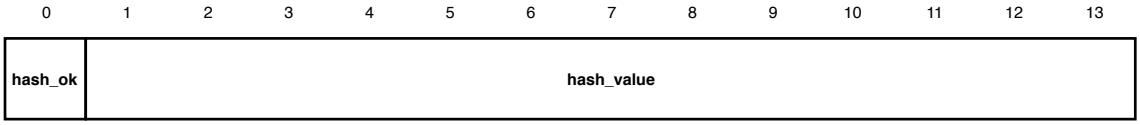
Obr. 4.7: Abstraktná schéma stavov a prechodov parser bloku P4 zreťazenia použitého vo verifikácii

Verifikácia overuje správnosť výpočtu a zapojenia externého objektu do zreťazenia P4. Toto zreťazenie je zobrazené na obrázku 4.8.



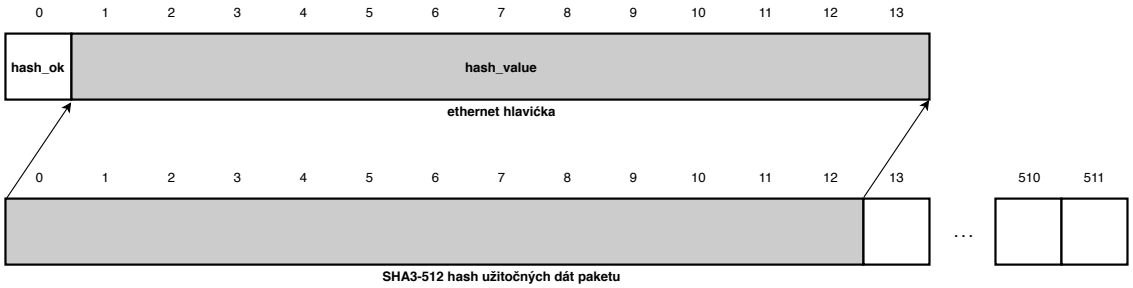
Obr. 4.8: Abstraktná bloková schéma umiestnenia externého objektu v P4 zreťazení

Keďže verifikácia komponenty FLU_WRAPPER_SHA3, popísaná v sekcii 4.3.3 overila správnosť výpočtu samotného hashu, v tejto verifikácii bola overovaná funkcionalita zapojenia v P4 zreťazení. Ako overenie použitia respektíve modifikácie hlavičky dátami z kryptografického externého objektu, bol samotný hash užitočných dát paketu respektíve jeho časť vložená do ethernet hlavičky každého paketu ako:



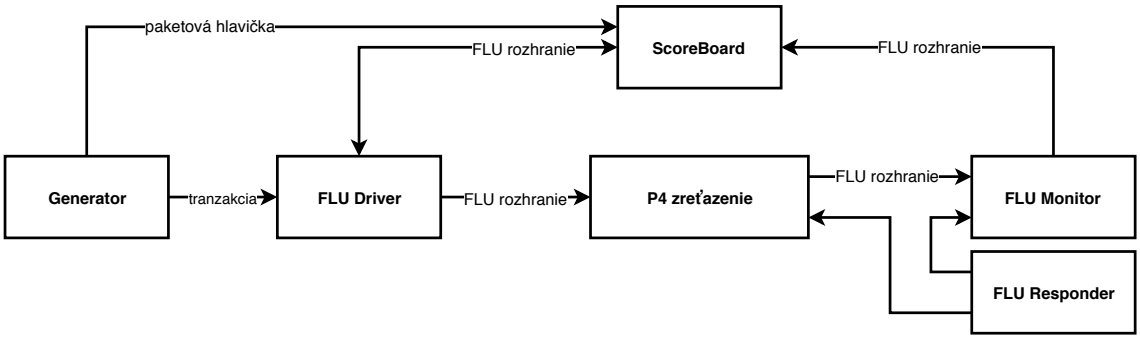
Obr. 4.9: Umiestnenie časti hashu do ethernet hlavičky

, kde prvý bajt obsahuje hodnotu `0x01` ak je hodnota hashu valídna a `0x00` ak sú užitočné dáta pre daný paket prázdne respektíve ich neobsahuje a tým pádom je hash nevalídny. Na obrázku 4.9 je táto hodnota pomenovaná ako `hash_ok`. Ďalších trinásť bajtov, pomenovaných ako `hash_value` obsahuje prvých trinásť bajtov hashu, ako zobrazuje obrázok 4.10.



Obr. 4.10: Umiestnenie časti hashu do ethernet hlavičky 2

Samotná schéma jednotlivých blokov verifikácie je zobrazené na obrázku 4.11.



Obr. 4.11: Abstraktná schéma blokov verifikácie P4 zreťazenia

Pre každý z piatich scenárov verifikácie bolo vyslaných 20 000 transakcií. Výpis H.1 v prílohe H popisuje nastavenia jednotlivých parametrov ako napríklad minimálnu a maximálnu veľkosť paketov, obmedzenia pre oneskorenia v a medzi transakciami na vysielacej strane, blok Driver, a na strane príjmu, blok Responder a Monitor. Výstup verifikácie je vo výpise H.2 v prílohe H, ktorý ukazuje úspešné ukončenie všetkých päť scenárov a teda úspešný prechod 100 000 transakcií.

5 Podpora kryptografických externých objektov

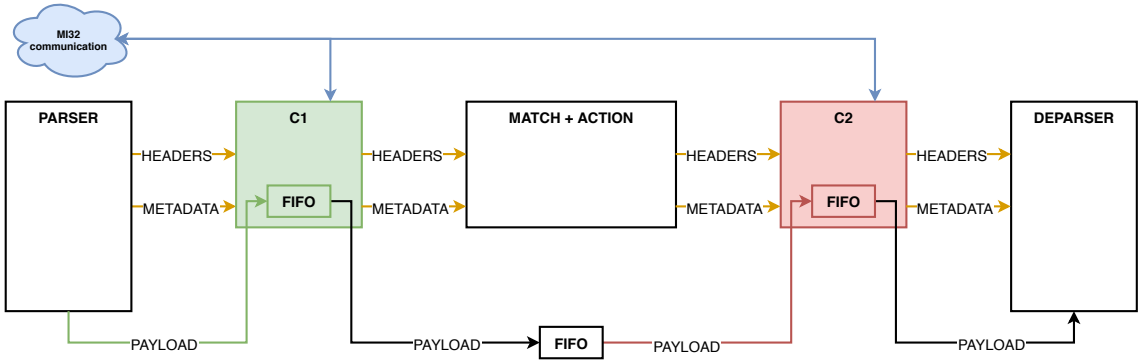
Táto kapitola popisuje samotnú implementáciu a rozšírenie kompilátoru o podporu kryptografických externých objektov.

5.1 Architektúra riešenia

Na základe výsledkov testovania zapojenia kryptografického externého objektu do P4 zreťazenia, popísaného v kapitole 4, bol zvolený nasledujúci návrh.

5.1.1 P4 zreťazenie

Vzhľadom na kompatibilitu s V1 modelom[1] pozostávajúcim z blokov `MyParser`, `MyVerifyChecksum`, `MyIngress`, `MyEgress`, `MyComputeChecksum` a `MyDeparser` bolo zvolené použitie kontrolných blokov C1 a C2 na umiestnenie kryptografických externých objektov. Architektúra P4 zreťazenia využívajúceho tieto kontrolné bloky je zobrazená na obrázku 5.1, kde kontrolný blok C1 je ekvivalentom MyVerifyChecksum kontrolného bloku a C2 MyComputeChecksum kontrolného bloku. Ich popis v jazyku P4 je zobrazený vo výpise 5.1.1.



Obr. 5.1: Umiestnenie kontrolných blokov v P4 zreťazení

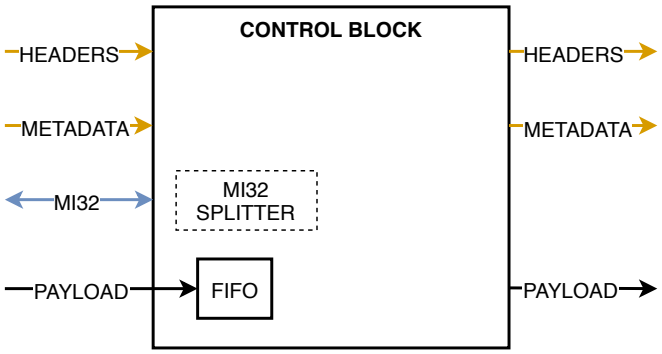
```
1 /*************************************************
2 ************   C1 control block   *************
3 *************************************************/
4 control MyVerifyChecksum(inout headers hdr,
```

[1] https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

```
5                                   inout metadata meta) {
6       apply {  }
7   }
8
9   /*************************************************
10  ************   C2 control block  *************
11  *************************************************/
12  control MyComputeChecksum(inout headers  hdr,
13                                  inout metadata meta) {
14      apply {  }
15  }
```

5.1.2 Kontrolné bloky

Keď že kryptografické externé objekty môžu spracovávať rôzne dáta, je nutné ich dostať na rozhranie kontrolných blokov. Toto rozhranie je zobrazené na obrázku 5.2. Bolo navrhnuté s ohľadom na nutnosť sprístupniť všetky potrebné dáta, ktoré by rôzne kryptografické externé objekty potrebovali spracovávať.

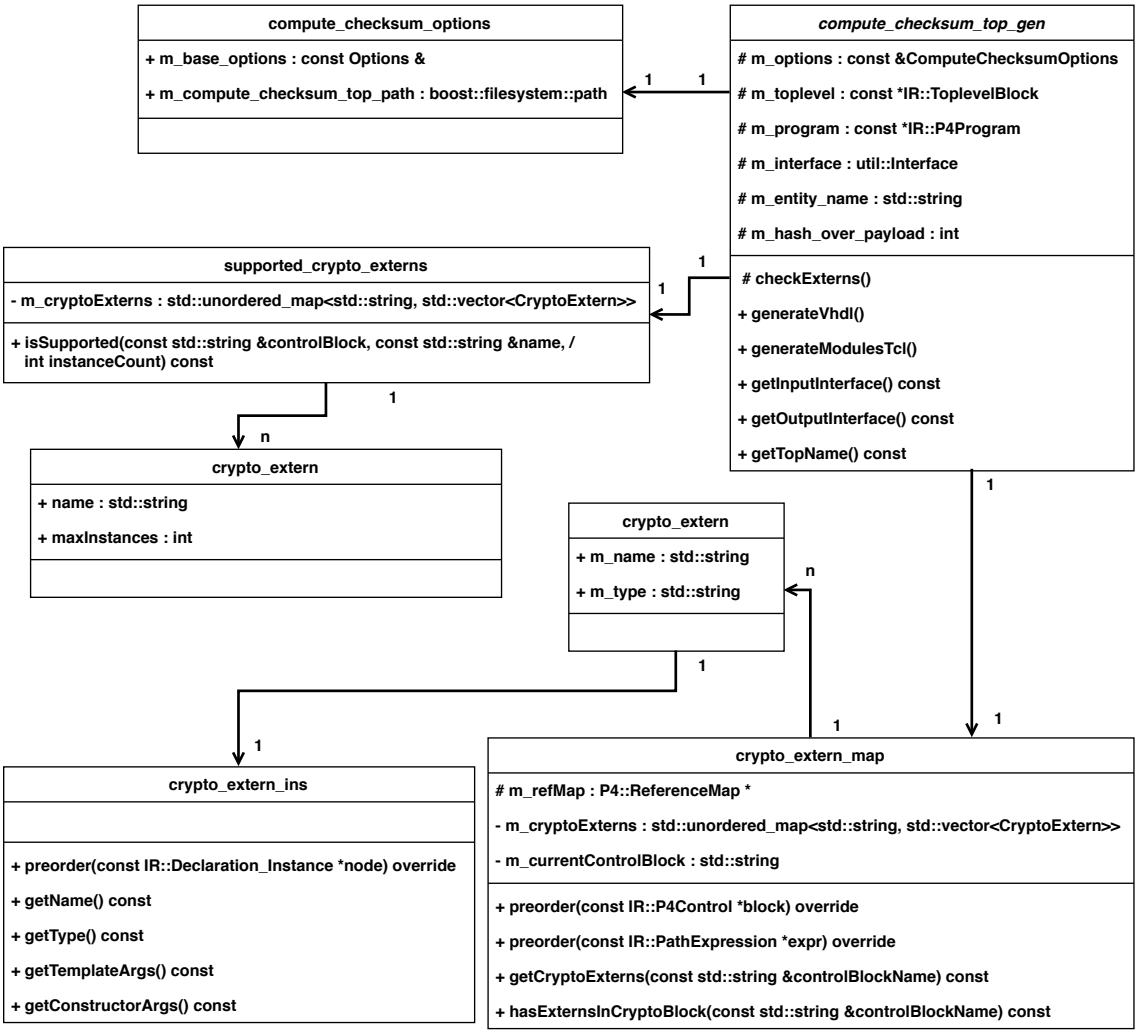


Obr. 5.2: Rozhranie kontrolného bloku

Rozhranie MI32 komunikácie slúži na sprístupnenie konfiguračného rozhrania pre jednotlivé kryptografické externé objekty. Vzhľadom na to, že jednotlivé kryptografické externé objekty môžu dáta aj modifikovať, definuje rozhranie kontrolného bloku vždy vstupné aj výstupné rozhranie pre jednotlivé dáta. Vstupným respektíve výstupným rozhraním `HEADERS` putujú oddelené hlavičky paketu. V `METADATA` rozhraní nájdeme rôzne potrebné dáta pre jednotlivé pakety, napríklad šifrovacie kľúče, identifikáciu vstupného rozhrania paketu a iné. Poslená časť `PAYLOAD` sprístupňuje samotné užitočné dáta paketu.

5.2 Úprava kompilátoru

V tejto sekcii je popísaná úprava kompilátoru pre generovanie kontrolného bloku C2, popísaného v sekcii 5.1, o podporu spracovávania a instancovania kryptografických externých objektov a podporu konkrétneho kryptografického externého objektu a to `hashOverpPayload`. Jednotlivé podskecie reprezentujú časti , ktoré bolo potrebné vytvoriť. Obrázok 5.3 zobrazuje diagram tried implementovaného rozšírenia.



Obr. 5.3: Diagram tried rozšírenia kompilátoru

5.2.1 Predlohy

Jedná sa o predlohy (angl.: template) VHDL zdrojových kódov, ktoré sa generujú a môžu využívať premenné a podmienky, na základe ktorých sa modifikuje samotný koncový zdrojový kód. Tieto premenné sú plnené pomocou obslužných metód, ktoré z predlohy generujú tento koncový zdrojový kód. Ich použitie je popísané v ďalších

častiach práce. Predloha pre jednotlive architektúry kontrolného bloku sa vytvárajú pre každý kryptografický externý objekt zvlášť, a preto budú popísané v časti podpory konkrétneho kryptografického externého objektu.

compute_checksum_top_entity_template

Predloha definujúca rozhranie kontrolného bloku C2, takzvanú VHDL entitu tejto komponenty. Keďže podpora metadát je v kompilátore v štádiu vývoja, je pre toto rozhranie spravená zatiaľ iba predpríprava. Predloha je zobrazená vo výpise I.1.

zdrojový súbor: compute_checksum_top_template.h
názov predlohy: compute_checksum_top_entity_template

compute_checksum_top_architecture_empty_template

V prípade, kedy v zdrojvom P4 kóde nie je použitý žiadny kryptografický externý blok v kontrolnom bloku C2, vygeneruje sa a použije sa prázdna architektúra (angl.: empty architecture) pre túto komponentu, a teda iba sa prepoja jej vstupy na výstupy. Túto predlohu zobrazuje výpis I.2.

zdrojový súbor: compute_checksum_top_template.h
názov predlohy: compute_checksum_top_architecture_empty_template

p4_top_template

Pre instancovanie kontrolného bloku C2 v P4 zreťazení bolo nutné upraviť predlohu tohto P4 zreťazenia a vytvoriť jednotlivé prepojenia. To vzniklo vložením komponenty kontrolného bloku C2 medzi časti Match+Action a Deparser časť a úpravou ich prepojenia tak, že výstup Match+Action časti je zapojený na vstup C2 kontrolného bloku, ktorého výstup je následne prepojený so vstupom časti Deparser. Výpis I.3 zobrazuje úpravu tejto predlohy použitím nástroja `git diff`.

zdrojový súbor: top_template.h
názov predlohy: top_architecture_full_template

5.2.2 Compute checksum options

Jedná sa o rozšírenie základných nastavení (angl.: options) pre kompilátor, ktoré uchovávajú potrebné údaje pre generovanie. Výpis 5.1 zobrazuje hlavičkový súbor a výpis 5.7 C++ zdrojový súbor.

Výpis 5.1: Hlavičkový súbor compute_checksum_options.h

```
#ifndef COMPUTE_CHECKSUM_OPTIONS_H_
#define COMPUTE_CHECKSUM_OPTIONS_H_

#include <string>
#include <boost/filesystem.hpp>
#include "util/options.h"

namespace compute_checksum
{
struct ComputeChecksumOptions {
  const Options &m_base_options;
  boost::filesystem::path m_compute_checksum_top_path;
  ComputeChecksumOptions(const Options &options);
};
} //compute_checksum
#endif // COMPUTE_CHECKSUM_OPTIONS_H_
```

Výpis 5.2: Zdrojový súbor compute_checksum_options.cpp

```
#include "compute_checksum_options.h"

using namespace compute_checksum;

ComputeChecksumOptions::ComputeChecksumOptions(const /
Options &options) : m_base_options(options)
{
  m_compute_checksum_top_path = m_base_options.m_GenDir;
  m_compute_checksum_top_path /= "compute_checksum";
}
```

Pozostáva z dvoch členských atribútov triedy a to `m_compute_checksum_top_path`, ktorý uchováva cestu k hlavnej zložke, kde sa bude generovať kód pre C2 kontrolný blok a základných nastavení v `m_base_options`.

5.2.3 Crypto extern inspector

Ide o inšpektora, ktorý sa púšťa nad každým `IR::Declaration_Instance` uzlom (angl.: node) pri spracovávaní v crypto extern mape, popísanej v nasledujúcej časti.

Z uzlu vracia jeho meno pomocou funkcie `getName`, typ funkciou `getType`, argumenty predlohy pomocou funkcie `getTemplateArgs` a argumenty konštruktoru pomocou `getConstructorArgs`. Výpis 5.3 zobrazuje hlavičkový a výpis 5.4 zdrojový súbor inšpektoru.

Výpis 5.3: Hlavičkový súbor crypto_extern_ins.h

```
1  #ifndef CRYPTO_EXTERN_INS_H_
2  #define CRYPTO_EXTERN_INS_H_
3
4  #include "match_action/inspectors/extern_ins.h"
5
6  namespace crypto
7  {
8  class CryptoExternIns : public hls::ExternInspector {
9    public:
10   bool preorder(const IR::Declaration_Instance *node) /
11   override;
12   const std::string &getName() const;
13   const std::string &getType() const;
14   const IR::Vector<IR::Type> *getTemplateArgs() const;
15   const IR::Vector<IR::Argument> *getConstructorArgs() /
16   const;
17 };
18 } // namespace crypto
19 #endif // CRYPTO_EXTERN_INS_H_
```

Výpis 5.4: Zdrojový súbor crypto_extern_ins.cpp

```
1  #include "crypto_extern_ins.h"
2
3  using namespace crypto;
4
5  bool CryptoExternIns::preorder(const /
6  IR::Declaration_Instance *node) {
7    m_externName = std::string(node->name.toString()/
8    .c_str());
9    m_externType = std::string(node->type->to/
10   <IR::Type_Name>()->path->name.toString().c_str());
```

```
11    return true;
12  }
13
14  const std::string &CryptoExternIns::getName() const {
15    return m_externName;
16  }
17
18  const std::string &CryptoExternIns::getType() const {
19    return m_externType;
20  }
21
22  const IR::Vector<IR::Type> *CryptoExternIns::/
23  getTemplateArgs() const {
24    return m_templateArgs;
25  }
26
27  const IR::Vector<IR::Argument> *CryptoExternIns::/
28  getConstructorArgs() const {
29    return m_ctorArgs;
30  }
```

5.2.4 Crypto extern map

Prechádza P4 zdrojový kód a vytvára mapu, do ktorej zaznamenáva kryptografické externé objekty, ktoré sa v tomto zdrojovom kóde nachádzajú. Zdrojové súbory sú zobrazené vo výpisoch 5.5 a 5.6. Metóda `preorder(const IR::P4Control* block)` sa aplikuje na konrolný blok C2 a teda „MyComputeChecksum“, kde zaznamená aktuálny kontrolný blok v ktorom sa nachádza. Je to predprípdava na pridanie C1 kontrolného bloku. Metóda `preorder(const IR::PathExpression *expr)` prechádza uzly a pridáva ich do mapy, kde pomocou konštruktoru `CryptoExtern(declaration)`, a teda použitím crypto extern inšpektora vytvára záznamy o kryptografických externých objektoch. Pomocou metódy `getCryptoExterns(const std::string &controlBlockName)` vracia kryptografické externé objekty konkrétneho kontrolného bloku.

Výpis 5.5: Hlavičkový súbor crypto_extern_map.h

```
1  #ifndef CRYPTO_EXTERN_MAP_H_
2  #define CRYPTO_EXTERN_MAP_H_
```

```
3
4  #include "ir/ir.h"
5  #include "frontends/common/resolveReferences/
6            referenceMap.h"
7
8  namespace util
9  {
10 struct CryptoExtern {
11   std::string m_name = "";
12   std::string m_type = "";
13   CryptoExtern(const IR::IDeclaration *declaration);
14 };
15
16 class CryptoExternMap : public Inspector {
17 public:
18     CryptoExternMap(P4::ReferenceMap *refMap);
19     bool preorder(const IR::P4Control *block) override;
20     bool preorder(const IR::PathExpression *expr) /
21     override;
22     const std::vector<CryptoExtern> & getCryptoExterns/
23     (const std::string &controlBlockName) const;
24     bool hasExternsInCryptoBlock(const std::string /
25     &controlBlockName) const;
26   protected:
27     P4::ReferenceMap *m_refMap;
28   private:
29     std::unordered_map<std::string, std::vector</
30     CryptoExtern>> m_cryptoExterns;
31     std::string m_currentControlBlock = "";
32 };
33 } //namespace util
34 #endif // CRYPTO_EXTERN_MAP_H_
```

Výpis 5.6: Zdrojový súbor crypto_extern_map.cpp

```
1  #include "crypto_extern_map.h"
2  #include "extern_map.h"
3  #include "crypto/crypto_extern_ins.h"
4
```

```
5  using namespace util;
6
7  CryptoExternMap::CryptoExternMap(P4::ReferenceMap *refMap)
8  : m_refMap(refMap){}
9
10 bool CryptoExternMap::preorder(const IR::P4Control* block)
11 {
12   if (block->name != "MyComputeChecksum") {
13     return false;
14   }
15   m_currentControlBlock = std::string(block->name/
16   .toString());
17   return true;
18 }
19
20 bool CryptoExternMap::preorder(const IR::PathExpression /
21 *expr) {
22   if (!ExternMap::isExternExpression(expr)) {
23     return true;
24   }
25   const IR::IDeclaration *declaration = m_refMap/
26   ->getDeclaration(expr->path);
27   m_cryptoExterns[m_currentControlBlock].push_back/
28   (CryptoExtern(declaration));
29   return false;
30 }
31
32 const std::vector<CryptoExtern> & CryptoExternMap::/
33 getCryptoExterns(const std::string &controlBlockName) /
34 const {
35   if (!hasExternsInCryptoBlock(controlBlockName)) {
36     throw std::runtime_error("No entry for given /
37     control block");
38   }
39   return m_cryptoExterns.at(controlBlockName);
40 }
41
42 bool CryptoExternMap::hasExternsInCryptoBlock(const /
43 std::string &controlBlockName) const {
```

```
44    return m_cryptoExterns.find(controlBlockName) != /
45    m_cryptoExterns.end();
46  }
47
48  CryptoExtern::CryptoExtern(const IR::IDeclaration /
49  *declaration)
50  {
51    crypto::CryptoExternIns ins;
52    declaration->getNode()->apply(ins);
53    m_name = ins.getName();
54    m_type = ins.getType();
55  }
```

5.2.5 Supported crypto externs

Slúži na uchovávanie podporovaných kryptografických externých objektov kompilátorom. Používa sa následne priamo v compute checksum top gen časti, kde je využívaná metóda `isSupported` na overenie, či je použitý kryptografický externý objekt kompilátorom podporovaný, a teda je možné ho použiť. Disponuje aj chybovou hláškou o nemožnosti použiť nepodporovaný kryptografický externý objekt ako je možné vidieť vo výpise 5.8, zobrazujúcom zdrojový súbor. Výpis 5.7 následne zobrazuje hlavičkový súbor. Je možné uchovávať informáciu, aký kryptografický externý objekt je podporovaný v akom konkrétnom kontrolnom bloku a koľko instancií je možné použiť.

Výpis 5.7: Hlavičkový súbor supported_crypto_externs.h

```
1  #ifndef SUPPORTED_CRYPTO_EXTERNS_H_
2  #define SUPPORTED_CRYPTO_EXTERNS_H_
3
4  #include <string>
5  #include <vector>
6  #include <unordered_map>
7
8  namespace crypto
9  {
10 struct CryptoExtern {
11   std::string name = "";
12   int maxInstances;
13   CryptoExtern(const std::string &name, /
```

```
14    const int maxInstances);
15  };
16
17  class SupportedCryptoExterns {
18    private:
19      std::unordered_map<std::string, /
20      std::vector<CryptoExtern>> m_cryptoExterns;
21    public:
22      SupportedCryptoExterns();
23      bool isSupported(const std::string &controlBlock, /
24      const std::string &name, int instanceCount) const;
25  };
26  } // namespace crypto
27  #endif // SUPPORTED_CRYPTO_EXTERNS_H_
```

Výpis 5.8: Zdrojový súbor supported_crypto_externs.cpp

```
1   #include "supported_crypto_externs.h"
2   #include "lib/exceptions.h"
3
4   using namespace crypto;
5
6   SupportedCryptoExterns::SupportedCryptoExterns() {
7   }
8   bool SupportedCryptoExterns::isSupported(const /
9   std::string &controlBlock, const std::string &name, /
10  int instanceCount) const {
11    if(m_cryptoExterns.find(controlBlock) != /
12    m_cryptoExterns.end()) {
13      for(crypto::CryptoExtern ext : m_cryptoExterns/
14      .at(controlBlock)) {
15        if (ext.name == name && ext.maxInstances >= /
16        instanceCount) {
17          return true;
18        }
19      else {
20        BUG("Using of crypto extern %1% in control block /
21        %2% is not supported", name, controlBlock);
22      }
```

```
23      }
24    }
25    return false;
26  }
27
28  CryptoExtern::CryptoExtern(const std::string &name, /
29  const int maxInstances) :
30    name(name),
31    maxInstances(maxInstances){}
```

5.2.6 Compute checksum top gen

Stará sa o samotnú obsluhu generovania kontrolného bloku C2 a jednotlivých kryptografických externých objektov. Používa vyššie popísané súčasti kompilátoru. Výpis 5.9 a I.5 zobrazujú zdrojové súbory.

Výpis 5.9: Hlavičkový súbor compute_checksum_top_gen.h

```
1  #ifndef _BACKENDS_VHDL_COMPUTE_CHECKSUM_TOP_GEN_H_
2  #define _BACKENDS_VHDL_COMPUTE_CHECKSUM_TOP_GEN_H_
3
4  #include "compute_checksum_options.h"
5  #include "util/interface.h"
6  #include "util/crypto_extern_map.h"
7  #include "crypto/supported_crypto_externs.h"
8
9  namespace compute_checksum {
10
11 class ComputeChecksumTopGen {
12   protected:
13     const ComputeChecksumOptions &m_options;
14     const IR::ToplevelBlock *m_toplevel;
15     const IR::P4Program *m_program;
16     util::Interface m_interface;
17     std::string m_entity_name;
18     std::string genEntity() const;
19     std::string genModulesTcl() const;
20     std::string genEmptyArch() const;
21     std::string genFullArch() const;
```

```
22      util::CryptoExternMap *m_cryptoExternMap;
23      crypto::SupportedCryptoExterns /
24      *m_supportedCryptoExterns;
25      int m_hash_over_payload;
26      void checkExterns();
27    public:
28      ComputeChecksumTopGen(const ComputeChecksumOptions& /
29      options, const IR::ToplevelBlock *toplevel, /
30      const util::Interface
31      &interface, util::CryptoExternMap *cryptoExternMap);
32      void generateVhdl();
33      void generateModulesTcl();
34      const util::Interface getInputInterface() const;
35      const util::Interface getOutputInterface() const;
36      std::string getTopName() const;
37  };
38
39  } // namespace compute_checksum
40
41  #endif // _BACKENDS_VHDL_COMPUTE_CHECKSUM_TOP_GEN_H_
```

Verejé metódy sú volané z nadradenej triedy. Pomocou `getTopName` je možné zistiť meno top entity kontrolného bloku C2. Metódy `getInputInterface` a `getOutputInterface` vracajú rozhranie kontrolného bloku. Vygenerovanie celého VHDL zdrojového kódu jednotlivých súčasťi a samotného kontrolného bloku je možné pomocou metódy `generateVhdl`.

Interné chránené premenné a metódy slúžia ako pomocné súčasti pri generovaní kontrolného bloku. Metóda `genEntity` generuje predlohu pre entitu kontrolného bloku. Pomocou metód `genEmptyArch` a `genFullArch` sa generuje predloha architektúry kontrolného bloku. Potrebnný súbor pre mapovanie jednotlivých zdrojových súborov Modules.tcl respektíve jeho predloha sa genruje pomocou metódy `genModulesTcl`. Generovanie špecifických predlôh funguje na princípe napĺňania premenných v predlohách. Pomocou metódy `checkExterns` sa overujú jednotlivé externé objekty použité v P4 zdrojovom kóde voči kryptografickým externým objektom podporovanými kompilátorom. Ostatné premenné nesú informácie o rozhraní hlavičiek, názve entity, P4 programe a nastavení kompilátoru.

5.2.7 Externý objekt HashOverPayload

Do kompilátoru bola pridaná podpora pre kryptografický externý objekt HashOverPayload, ktorý vypočítava SHA3-512 hash užitočných dát paketu a vkladá jeho hodnotu do ethernetovej hlavičky tak, ako je popísané v kapitole 4. Pre túto podporu bolo nutné vytvoriť predlohu architektúry, ktorá správne instancuje a zapája FLU_WRAPPER_SHA3 komponentu a synchronizuje hlavičky a vloženie hodnoty hashu do ethernetovej cieľovej adresy. Táto architektúra je zobrazená vo výpise I.4. Pomocou podmienenia parametru:

```
{% if hash_over_payload == 1 %}
```

je možné generovať architektúru pre tento konkrétny kryptografický externý blok.

zdrojový súbor:	compute_checksum_top_template.h
názov predlohy:	compute_checksum_top_architecture_full_template

Taktiež bolo nutné pridať tento kryptografický externý objekt do štruktúry podporovaných kryptografických externých objektov kompilátorom, a teda rozšíriť konštruktor v zdrojovom súbore supported_crypto_externs.cpp, čo zobrazuje výpis 5.10.

Výpis 5.10: Rozšírený zdrojový súbor supported_crypto_externs.cpp

```
1   SupportedCryptoExterns::SupportedCryptoExterns() {
2 +   m_cryptoExterns["MyComputeChecksum"].push_back(/
3     CryptoExtern("HashOverPayload", 1));
4   }
```

Správny zápis a použitie kryptografického externého objektu HashOverPayload v P4 zdrojovom kóde zobrazuje výpis 5.11.

Výpis 5.11: Časť P4 zdrojového kódu používajúca hash_over_payload objekt

```
1 extern HashOverPayload {
2     // Constructor
3     HashOverPayload();
4
5     void computeAndAddToEthHeader();
6 }
7
8 .
```

```
9  .
10 .
11
12 control MyComputeChecksum(inout headers  hdr, /
13 inout metadata meta) {
14
15     HashOverPayload() hash_over_payload;
16
17     apply {
18         hash_over_payload.computeAndAddToEthHeader();
19     }
20 }
```

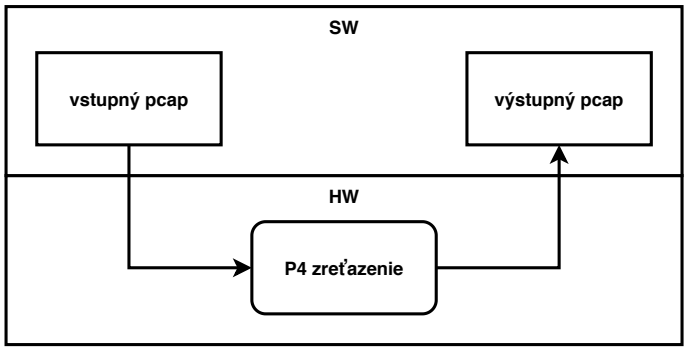
Objekt typu `HashOverPayload` s názvom `hash_over_payload` volá jedinú definovanú metódu a to `computeAndAddToEthHeader()`.

Testovanie

Overenie správnej fukncionality bolo zastrešované vytvorením verifikácie používajúcej verifikačné prostredie CESNETu a V1 model. Táto verifikácia simuluje správanie sa reálneho zapojenia na základe prekladu z P4 zdrojového súboru. K verifikácii je potrebný vstupný a očakávaný výstupný pcap a konfiguračné pravidlá tabuliek.

Pre úplnosť testovania bola overená funkcionalita v reálnom hardvéri. P4 zdrojový kód bol preložený do firmwaru pomocou upraveného kompilátoru pre sieťovú kartu NFB-200G2QL, popísanú v sekcii 2.4, na ktorej prebehol test. Abstraktná schéma toku dát počas testovania je zobrazené na obrázku 5.4. Do karty boli na vstupné rozhranie vyslané dáta pomocou softvérového nástroja. Karta tieto dáta spracovala, a teda pakety prešli P4 zreťazením obsahujúcim a používajúcim kryptografický externý objekt HashOverPayload. Karta tieto pakety následne vyslala na výstupné rozhranie, kde boli pomocou softvérového nástroja zachytené. Zachytené dáta a teda výstupný pcap bol následne porovnaný s predgenerovaným očakávaným výsledkom a zhodoval sa. Vygenerovaný firmware teda funguje správne.

V tabuľke 5.1 je zobrazený výstup zabraných zdrojov preloženého firmwaru, kde je možné vidieť, že sa tam jednotlivé komponenty a celé P4 zreťazenie napasovalo s využitím neveľkej časti čipu.



Obr. 5.4: Abstraktná schéma toku dát počas testovania

Tab. 5.1: Report zabraných zdrojov

Typ zdroja	**Použité zdroje**	**Dostupné zdroje**	**Použité zdroje** [%]
CLB LUTs	246031	788160	31,34
CLB Registers	261909	1576320	16,62
CARRY8	2019	98520	2,05

Testovanie celkovej priepustnosti dopadlo podla očakávaní. Je zobrazené v tabuľke 5.2. Výsledná priepustnosť je nižšia ako maximálna, čo je zapríčinené použitím existujúcej implementácie komponenty SHA3. Neefektivita je spôsobená jej implementáciou a samotným malým vstupným rozhraním. Je schopná spracovávať iba šesťdesiatštyri bitov za takt na vstupnom rozhraní a následne podobný počet taktov ako boli spracovávané vstupy prebieha interný výpočet do doby, než je vrátený výsledok na výstupnom rozhraní. Rozhranie kontrolného bloku C2 ako aj celé P4 zreťazenie však dosahuje plnú priepustnosť.

Tab. 5.2: Výsledky testu priepustnosti

Veľkosť [B]	**Data rate [Gb/s]**	**Link rate [Gb/s]**	**Prijaté** [%]	**Zahodené** [%]	**Priepustnosť [Gb/s]**
64	94,324	123,628	9,830	90,169	12,152
128	97,262	115,618	12,206	87,793	14,112
256	102,773	110,775	14,062	85,937	15,577
512	103,957	108,017	17,231	82,768	18,612
1024	104,649	106,692	18,155	81,844	19,369

6 Záver

Úlohou tejto diplomovej práce bolo zoznámiť sa a popísať samotný jazyk P4 a jeho podporu externých objektov, zoznámiť sa s prostriedkami pre mapovanie externých objektov na FPGA čip, navrhnúť a popísať samotnú implementáciu externých objektov do P4 zreťazenia a teda mapovania na FPGA čip. Vytvoriť simuláciu podpory vybraného kryptografického externého objektu ako overenie navrhnutého riešenia. Navrhnuté riešenie implementovať ako rozšírenie kompilátoru. Následne pridať podporu konkrétneho kryptografického externého objektu za použitia existujúcej implementácie SHA3.

V úvode teoretickej časti práca popisuje samotný jazyk P4, jeho prvky, porovnáva jednotlivé revízie a jeho podporu externých objektov. Zoznamuje čitatela s použitými technológiami ako programovateľnými hradlovými polami, HDL jazykmi, produktom Netcope P4 a vysoko-rýchlostnými sieťovými kartami s FPGA čipom alebo verifikáciami a SystemVerilog DPI.

V ďalšej časti sa venuje rozboru a návrhu mapovania externých objektov do P4 zreťazenia a teda na samotný FPGA čip, pričom popisuje samotný P4-VHDL kompilátor a architektúru P4 zreťazenia.

Praktická časť sa na začiatku venuje popisu vytvorenia verifikácie P4 zreťazenia s kryptografickým externým objektom a jednotlivé kroky potrebné na jej vytvorenie ako overenie funkčnosti navrhnutého riešenia. Na začiatku popisuje samotný vybraný kryptografický externý objekt a to hashovaciu funkciu SHA3-512. Popisu výber IP Coru a referenčnej implementácie tejto funkcie v jazyku C. Potrebné úpravy, ktoré bolo nutné spraviť. Simuláciu samotného IP Coru. Venuje sa prispôsobeniu IP Coru na prácu s dátami pomocou FLU rozhrania a teda vytvoreniu FLU obálky v jazyku VHDL na komponentu hashovacej funkcie pre vytvorenie tejto kompatibility. Zobrazuje výsledky simulácie a overenie funkcionality pomocou testovacích vektorov hashovacej funkcie, výsledky syntézy celej obálky s IP Corom a vytvorenie a samotnú verifikáciu. Popisuje vytvorenie P4 zreťazenia s kryptografickým externým objektom, jeho zapojenie a to, ako pracuje, vytvorenie verifikácie a jej výsledky.

Následne popisuje implementáciu rozšírenia kompilátoru o podporu kontrolného bloku C2 a teda podporu kryptografických externých objektov. Popisuje jednotlivé časti, ktoré boli implementované. Na začiatku sa venuje predlohám pre generovanie VHDL zdrojových kódov. Popisuje implementáciu inšpektora a crypto extern mapy, pomocou ktorých sa spracováva P4 zdrojový kód a v ňom použité kryptografické externé objekty. Ďalej sa venuje popisu podporovaných kryptogrfických externých objektov kompilátorom ako aj definícii nastavení kompilátoru a to compute checksum options. V závere popisuje compute checksum top gen časť, ktorá má na starosti samotné generovanie a zapojenie kontrolného bloku C2 do P4 zreťazenia a jednot-

livých kryptografických externých objektov v ňom. Ďalej sa venuje implementácii podpory kryptografického externého objektu HashOverPayload, jeho použitiu a následnému testu celej implementácie a zhodnoteniu výsledkov.

V tejto práci sa podarilo navrhnúť, otestovať návrh a následne implementovať rozšírenie kompilátoru P4-VHDL o podporu kryptografických externých objektov, konkrétne o kontrolný blok C2, ktorý tieto kryptografické externé objekty instancuje. V časti návrhu je popísané aj umiestnenie a rozhranie kontrolného bloku C1. Ďalej bol implementovaný jeden kryptografický externý objekt a to HashOverPayload, pomocou ktorého bola celá implementácia verifikovaná a následne testovaná v hardvéri. Samotná implementácia myslela a zahŕňa všetky potrebné súčasti na spracovávanie kryptografických externých objektov a jednoduché pridávanie podpory ďalších.

Možným rozšírením tejto práce je implementácia kontrolného bloku C1 prípadne rozšírenie o ďalšie kryptografické externé objekty.

V praxi je možné dané riešenie využiť pre jednoduchý vývoj kryptografických akceleračných aplikácii pre vysoko-rýchlostné sieťové karty pridaním kryptografických externých objektov pre šifrovanie dát, HMAC alebo digitálny podpis.

Literatúra

[1] BOSSHART, P.; DALY, D.; GIBB, G.; aj.: *P4: Programming Protocol-Independent Packet Processors. ACM SIGCOMM Computer Communication Review*, ročník 3, č. 44, 2014: s. 87–95, ISSN 0146-4833

[2] *P4 16 Language Specification*. Version 1.2.0-rc. The P4 Language Consortium, 2019. Dostupné z: `https://p4.org/p4-spec/docs/P4-16-v1.2.0-rc.pdf`

[3] CÍBIK, Peter. *Návrh a implementace šifry Twofish na síťové kartě FPGA*. Brno, 2018, 93 s. Bakalárska práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedúci práce: Ing. David Smékal

[4] Programovatelná logika II: FPGA. *Abclinuxu* [online]. [cit. 2019-10-23]. Dostupné z: `http://www.abclinuxu.cz/blog/digital_design/2013/1/programovatelna-logika-ii-fpga`

[5] PINKER, Jiří a Martin POUPA. *Číslicové systémy a jazyk VHDL*. Praha: BEN - technická literatura, 2006. ISBN 80-7300-198-5.

[6] KUBÍČEK, Michal. *Úvod do problematiky obvodú FPGA pro integrovanou výuku VUT a VŠB-TUO* [online]. Brno, 2014 [cit. 2019-10-24]. ISBN 978-80-214-5069-1. Dostupné z: `https://www.vutbr.cz/studium/ects-katalog/detail-predmetu?apid=149854`

[7] *IEEE Standard VHDL Language Reference Manual: Std 1076-2008*. IEEE, 2009. e-ISBN 978-0-7381-6853-1. Dostupné také z: `http://ieeexplore.ieee.org/document/4772740/`

[8] IEEE *Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language*. IEEE, 2017.

[9] Šimková, M.: *Hardwarově akcelerovaná funkční verifikace*. Diplomová práce, Vysokéučení technické v Brně, Fakulta informačních technologií, 2011. Dostupné z: `https://www.fit.vut.cz/study/thesis-file/9900/9900.pdf`

[10] Netcope products: *Netcope P4* [online]. [cit. 2019-11-04]. Dostupné z: `https://www.netcope.com/en/products/netcopep4`

[11] NETCOPE P4: *Product Brief* [online]. [cit. 2019-11-04]. Dostupné z: `https://www.netcope.com/en/products/netcopep4`

[12] Netcope products: *P4 to VHDL* [online]. [cit. 2019-11-04]. Dostupné z: https://www.netcope.com/en/products/p4-to-vhdl

[13] Netcope products: *Netcope FPGA boards (NFB)* [online]. [cit. 2019-11-11]. Dostupné z: https://www.netcope.com/en/products/fpga-boards

[14] Silicom products: *fb4CGg3@VU FPGA Card* [online]. [cit. 2019-11-11]. Dostupné z: https://www.silicom-usa.com/pr/fpga-based-cards/100-gigabit-fpga-cards/fb4cgg3vu-100-gigabit-xilinx-virtex-ultrascale/

[15] Intel: *Intel® FPGA Programmable Acceleration Card N3000* [online]. [cit. 2019-11-05]. Dostupné z: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-n3000/overview.html

[16] NETCOPE TECHNOLOGIES a.s. Netcope Development Kit: *Firmware developers manual*, 2018.

[17] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andrew Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer, 2006. ISBN: 0387-25556-7

[18] CABAL, Jakub. *Jak psát verifikace*. Brno: CESNET. Prezentácia.

[19] FIPS PUB 202: *SHA-3 Standard - Permutation-Based Hash and Extendable-Output Functions*. National Institute of Standards and Technology, 2015. Dostupné z: https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions

[20] P4c. *Github: p4lang/p4c* [online]. [cit. 2020-02-25]. Dostupné z: https://github.com/p4lang/p4c

Zoznam symbolov, veličín a skratiek

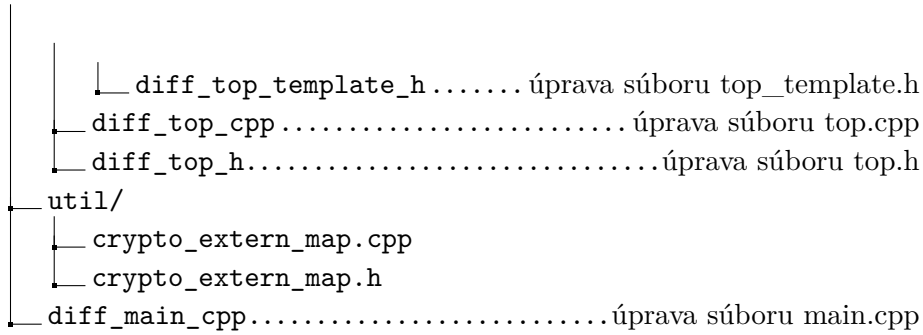
ABV	Assertion–Based Verification – verifikácia založená na výrokoch
arch	architektúra
ASIC	Application Specific Integrated Circuit
b	bit
CLB	Configurable Logic Block – programovateľný logický blok
CLK	Clock – hodiny
DPI	Direct Programming Interface
DUT	Design Under Test – testovaný návrh
Extern	Extern – externý objekt
FasS	Firmware as a Service – firmware ako služba
FPGA	Field Programable Gate Array – programovateľné hradlové pole
FIFO	First-In First-Out – prvé-dnu prvé-von, buffer na dáta
FIPS	Federal Information Processing Standards
FLU	FrameLinkUnaligned – protokol pre široké datové zbernice
FSM	Finite State Machine – konečný stavový automat
Gb/s	Gigabit za sekundu
HDL	Hardware Description Language – jazyk na popis hardwaru
HLS	High–Level Synthesis – nástroj na syntézu návrhu v jazyku C do RTL dizajnu
IBUF	Input BUFfer
IOB	Input-Output Block – vstupne-výstupný blok
IP	Intellectual Property
LUT	Look-Up Table – vyhľadávacia tabuľka
MHz	MegaHertz
NFB	Netcope Fpga Board – Netcope sieťová karta s FPGA čipom
NIST	National Institute of Standards and Technology
NP4	Netcope P4
ns	nanosecond – nanosekunda
OBUF	Output BUFfer
OOP	Object-Oriented Programming – objektovo orientované programovanie
PAC	Intel® Programmable Acceleration Card
PSA	Portable Switch Architecture
P4	Programming Protocol-independent Packet Processors
RTL	Register-Transfer Level
SHA	Secure Hash Algorithm
UUT	Unit Under Test – testovaná jednotka

VHDL	VHSIC Hardware Description Language
XKCP	The eXtended Keccak Code Package

Zoznam príloh

A Obsah digitálnej prílohy

- `/` — koreňový adresár
  - `crypto_git/` — SW a HW implementácie hashovacej funkcie
    - `sw/` — upravená SW implementácia (XKCP) SHA3 používaná vo verifikácii
    - `vhdl/` — hardverové komponenty
      - `sha3_keccak/` — upravený IP Core SHA3
        - `high_throughput_core/`
          - `flu_wrapper/` — FLU obálka pre IP Core
            - `comp/` — interné komponenty FLU obálky
            - `sim/` — simulácia FLU obálky
            - `synth/` — syntéza FLU obálky
            - `ver/` — verifikácia FLU obálky
            - `flu_wrapper_sha3_arch_full.vhd` — zdrojový kód architektúry FLU obálky
            - `flu_wrapper_sha3_ent.vhd` — zdrojový kód entity FLU obálky
  - `fw/` — firmware a pcapy z testovania
    - `hash_over_payload/`
      - `p4/` — P4 zdrojový kód
      - `test/` — pcapy testovania
        - `expected_out.pcap` — očakávaný výstup
        - `in.pcap` — vstup
        - `out.pcap` — výstup
      - `np4-200g2ql_p4.nfw` — firmware
  - `p4base/` — zdrojové kódy rozšírenia P4-VHDL kompilátoru
    - `compiler/`
      - `p4-16vhdl/`
        - `src/`
          - `compute_checksum/`
            - `templates` — predlohy VHDL zdrojových kódov
              - `compute_checksum_top_template.h`
              - `modules_tcl_template.h`
            - `compute_checksum_options.cpp`
            - `compute_checksum_options.h`
            - `compute_checksum_top_gen.cpp`
            - `compute_checksum_top_gen.h`
          - `crypto/`
            - `crypto_extern_ins.cpp`
            - `crypto_extern_ins.h`
            - `supported_crypto_externs.cpp`
            - `supported_crypto_externs.h`
          - `midend/`
            - `diff_midend_cpp` — úprava súboru midend.cpp
            - `diff_midend_h` — úprava súboru midend.h
          - `top/`
            - `templates/` — predlohy VHDL zdrojových kódov


diff_top_template_h ..... úprava súboru top_template.h
diff_top_cpp ..... úprava súboru top.cpp
diff_top_h ..... úprava súboru top.h
util/
crypto_extern_map.cpp
crypto_extern_map.h
diff_main_cpp ..... úprava súboru main.cpp

B Upravený zdrojový kód modulu `padder1`

Výpis B.1: Upravený zdrojový súbor `padder1.v`

```
1  module padder1(in, byte_num, out);
2
3      parameter IS_SHA = 0;
4
5      input       [63:0] in;
6      input       [2:0]  byte_num;
7      output reg  [63:0] out;
8
9      generate
10     if (IS_SHA == 1)
11       always @ (*)
12         case (byte_num)
13           0: out =              64'h0600000000000000;
14           1: out = {in[63:56], 56'h06000000000000};
15           2: out = {in[63:48], 48'h060000000000};
16           3: out = {in[63:40], 40'h0600000000};
17           4: out = {in[63:32], 32'h06000000};
18           5: out = {in[63:24], 24'h060000};
19           6: out = {in[63:16], 16'h0600};
20           7: out = {in[63:8],   8'h06};
21         endcase
22     else
23       always @ (*)
24         case (byte_num)
25           0: out =              64'h0100000000000000;
26           1: out = {in[63:56], 56'h01000000000000};
27           2: out = {in[63:48], 48'h010000000000};
28           3: out = {in[63:40], 40'h0100000000};
29           4: out = {in[63:32], 32'h01000000};
30           5: out = {in[63:24], 24'h010000};
31           6: out = {in[63:16], 16'h0100};
32           7: out = {in[63:8],   8'h01};
33         endcase
34     endgenerate
35 endmodule
```

C Upravený zdrojový kód funkcie FIPS202_SHA3_512

Výpis C.1: Upravený zdrojový súbor `Keccak-more-compact.c`

```
#include "dpiheader.h"

#define FOR(i,n) for(i=0; i<n; ++i)
typedef unsigned char u8;
typedef char s8;
typedef uint64_t u64;
typedef unsigned int ui;

...

void FIPS202_SHA3_512(const s8 *in, u64 inLen, u8 *out) {
    #ifdef DEBUG
        printf("Hi, this is FIPS202_SHA3_512 function\n");
        printf("Input: ");
        for (int i = 0; i<inLen; i++){
            printf("%c", in[i]);
        }
        printf("\nInput length: %d\n", inLen);
        Keccak(576, 1024,(unsigned char*) in, inLen,
               0x06, out, 64);
        printf("Output: ");
        for (int i = 0; i < 64; i++) {
            printf("%02x", out[i]);
        }
        printf("\n");
    #else
        Keccak(576, 1024, (unsigned char*) in, inLen,
               0x06, out, 64);
    #endif
}

...
```

D Výstup simulácie IP Coru – transcript

Výpis D.1: Finálna časť výstupu simulácie IP Coru

```
# [SHA3] - IS_SHA = 1
# [SHA3] input = The quick brown fox jumps/
                 over the lazy dog
# Check correct
# [SHA3] input = The quick brown fox jumps/
                 over the lazy dog.
# Check correct
# [SHA3] input = h'A1A2A3A4A5
# Check correct
# [SHA3] input = '' (empty string)
# Check correct
# [SHA3] input = h'EFCDAB9078563412EFCDAB9078563/
                   412EFCDAB9078563412EFCDAB9078/
                   563412EFCDAB9078563412EFCDAB9/
                   078563412EFCDAB9078563412EFCD/
                   AB9078563412
# Check correct
# [SHA3] input = h'EFCDAB9078563412EFCDAB9078563/
                 412EFCDAB9078563412EFCDAB907856/
                 3412EFCDAB9078563412EFCDAB90785/
                 63412EFCDAB9078563412EFCDAB9078/
                 563412EFCDAB9078563412EFCDAB907/
                 8563412EFCDAB9078563412EFCDAB90/
                 78563412EFCDAB9078563412EFCDAB9/
                 078563412EFCDAB9078563412EFCDAB/
                 9078563412EFCDAB9078563412EFCDA/
                 B907856
# Check correct
# Simulation finishes correctly!
# ** Note: $finish    : test_keccak.sv(339)
#    Time: 3980 ns  Iteration: 0  Instance: /test_keccak
# End time: 09:38:45 on Nov 13,2019, Elapsed time: 0:00:00
# Errors: 0, Warnings: 1
```

E Rozhranie FLU obálky pre IP Core

Výpis E.1: Entita obálky IP Coru

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.math_pack.all;
4
5  entity FLU_WRAPPER_SHA3 is
6      generic (
7          -- FLU data width
8          DATA_WIDTH          : integer := 512
9      );
10     port (
11         CLK                 : in  std_logic;
12         RESET               : in  std_logic;
13
14         RX_DATA             : in /
15         std_logic_vector((DATA_WIDTH - 1) downto 0);
16         RX_SOP              : in std_logic;
17         RX_SOP_POS          : in /
18         std_logic_vector((log2(DATA_WIDTH / 64) - 1)/
19         downto 0);
20         RX_EOP              : in std_logic;
21         RX_EOP_POS          : in /
22         std_logic_vector((log2(DATA_WIDTH / 8) - 1)/
23         downto 0);
24         RX_SRC_RDY          : in std_logic;
25         RX_DST_RDY          : out std_logic;
26
27         HASH_DATA           : out /
28         std_logic_vector((512 - 1) downto 0);
29         HASH_OK             : out std_logic;
30         HASH_VLD            : out std_logic;
31         HASH_DST_RDY        : in std_logic
32     );
33 end entity;
```

F Simulácia FLU obálky IP Coru

Výpis F.1: Konzolový výstup simulácie FLU obálky IP Coru

```
# * FLU_WRAPPER_SHA3 simulation start now *
# [SHA3] input = h'EFCDAB9078563412EFCDAB9078563412EFCDAB/
                  9078563412EFCDAB9078563412EFCDAB907856/
                  3412EFCDAB9078563412EFCDAB9078563412EF/
                  CDAB9078563412
# Check correct
# * TEST VECTORS simulation *
# from: https://csrc.nist.gov/Projects/Cryptographic-
/Algorithm-Validation-Program/Secure-Hashing#sha3vsha3vss
# CAVS 19.0
# SHA3-512 ShortMsg information for SHA3AllBytes1-28-16
# Length values represented in bits
# Generated on Thu Jan 28 13:32:47 2016
# [SHA3] input = h'6e0c65ee0943e34d9bbd27a8547690f2291f5a/
                  86d713c2be258e6ac16919fe9c4d491895d3a9/
                  61bb97f5fac255891a0eaa18f80e1fa1ebcb63/
                  9fcfc1
# Check correct
# [SHA3] input = h'bcc9849da4091d0edfe908e7c3386b0cadadb2/
                  859829c9dfee3d8ecf9dec86196eb2ceb093c5/
                  551f7e9a4927faabcfaa7478f7c899cbef4727/
                  417738fc06
# Check correct
# [SHA3] input = h'56ac4f6845a451dac3e8886f97f7024b64b1b1/
                  e9c5181c059b5755b9a6042be653a2a0d5d56a/
                  9e1e774be5c9312f48b4798019345beae2ffcc/
                  63554a3c69862e
# Check correct
# * Simulation finish correctly! *
# ** Note: $finish    : test_wrapper_flu_sha3.sv(207)
#    Time: 2170 ns  Iteration: 0  Instance: /testbench
# End time: 10:21:59 on Dec 03,2019, Elapsed time: 0:00:01
# Errors: 0, Warnings: 2
```

G Verifikácia komponenty FLU_WRAPPER_SHA3

Výpis G.1: Nastavenie parametrov verifikácie

```
// -------- CLOCKS AND RESETS ----------------------
parameter CLK_PERIOD = 10ns;
parameter RESET_TIME = 10*CLK_PERIOD;

// -------- DUT PARAMETERS -------------------------
// Number of lanes
parameter META_WIDTH = 120;
parameter DATA_WIDTH = 512;
parameter EOP_POS_WIDTH = log2(DATA_WIDTH/8);
parameter SOP_POS_WIDTH = 3;

// -------- FL TRANSACTION FORMAT ------------------
// FL packet size
int PACKET_SIZE_MIN[] = '{60  ,60  ,60, 60, 60};
int PACKET_SIZE_MAX[] = '{2048,2048,60, 60, 2048};

// -------- DRIVER PARAMETERS ----------------------
// Delay between transactions limits
parameter DRIVER_START_POS_LOW = 0;
parameter DRIVER_START_POS_HIGH = 2**SOP_POS_WIDTH-1;
// Delay enable/disalbe inside transaction weight
int DRIVER_INSIDE_DELAYEN_WT[] = '{1,1,1,1,1};
int DRIVER_INSIDE_DELAYDIS_WT[] = '{3,1,3,3,3};
// Delay inside transaction limits
int DRIVER_INSIDE_DELAYLOW[] = '{0,1,1,0,0};
int DRIVER_INSIDE_DELAYHIGH[] = '{0,2,2,0,0};

// -------- MONITOR PARAMETERS ---------------------
// Delay enable/disable between transactions weight
int MONITOR_DELAYEN_WT[] = '{1,1,1,1,1};
int MONITOR_DELAYDIS_WT[] = '{3,1,1,1,1};
// Delay between transactions limits
int MONITOR_DELAYLOW[] = '{0,1,1,0,0};
int MONITOR_DELAYHIGH[] = '{0,3,3,0,3};
```

```
35
36 // -------- TEST PARAMETERS ------------------------
37 // Count of transactions to generate
38 parameter TRANSACTION_COUNT = 20_000;
```

Výpis G.2: Konzolový výstup verifikácie

```
1  # ################################## #
2  #  FLU_WRAPPER_SHA3 verification   #
3  # ################################## #
4
5  # ######### TEST CASE 0 ############
6  # ----------------------------------
7  # --  TRANSACTION TABLE
8  # ----------------------------------
9  # Size:           0
10 # Items added:       20000
11 # Items removed:       20000
12 # ----------------------------------
13
14 # ######### TEST CASE 1 ############
15 # ----------------------------------
16 # --  TRANSACTION TABLE
17 # ----------------------------------
18 # Size:           0
19 # Items added:       20000
20 # Items removed:       20000
21 # ----------------------------------
22
23 # ######### TEST CASE 2 ############
24 # ----------------------------------
25 # --  TRANSACTION TABLE
26 # ----------------------------------
27 # Size:           0
28 # Items added:       20000
29 # Items removed:       20000
30 # ----------------------------------
31
32 # ######### TEST CASE 3 ############
```

```
33 # ----------------------------------
34 # --  TRANSACTION TABLE
35 # ----------------------------------
36 # Size:           0
37 # Items added:       20000
38 # Items removed:       20000
39 # ----------------------------------
40
41 # ######### TEST CASE 4 ############
42 # ----------------------------------
43 # --  TRANSACTION TABLE
44 # ----------------------------------
45 # Size:           0
46 # Items added:       20000
47 # Items removed:       20000
48 # ----------------------------------
```

H Verifikácia P4 zreťazenia s externým objektom

Výpis H.1: Nastavenie parametrov verifikácie

```
// -------- CLOCKS AND RESETS ----------------------
parameter CLK_PERIOD = 10ns;
parameter RESET_TIME = 10*CLK_PERIOD;

// -------- DUT PARAMETERS -------------------------
// Number of lanes
parameter META_WIDTH = 160;
parameter DATA_WIDTH = 512;
parameter EOP_POS_WIDTH = log2(DATA_WIDTH/8);
parameter SOP_POS_WIDTH = 3;

// -------- FL TRANSACTION FORMAT ------------------
// FL packet size
int PACKET_SIZE_MIN[] = '{60  ,60  ,60, 60, 60};
int PACKET_SIZE_MAX[] = '{2048,2048,60, 60, 2048};

// -------- DRIVER PARAMETERS ----------------------
// Delay between transactions limits
parameter DRIVER_START_POS_LOW = 0;
parameter DRIVER_START_POS_HIGH = 2**SOP_POS_WIDTH-1;
// Delay enable/disalbe inside transaction weight
int DRIVER_INSIDE_DELAYEN_WT[] = '{1,1,1,1,1};
int DRIVER_INSIDE_DELAYDIS_WT[] = '{3,1,3,3,3};
// Delay inside transaction limits
int DRIVER_INSIDE_DELAYLOW[] = '{0,1,1,0,0};
int DRIVER_INSIDE_DELAYHIGH[] = '{0,2,2,0,0};

// -------- MONITOR PARAMETERS ---------------------
// Delay enable/disable between transactions weight
int MONITOR_DELAYEN_WT[] = '{1,1,1,1,1};
int MONITOR_DELAYDIS_WT[] = '{3,1,1,1,1};
// Delay between transactions limits
int MONITOR_DELAYLOW[] = '{0,1,1,0,0};
int MONITOR_DELAYHIGH[] = '{0,3,3,0,3};
```

```
35
36 // Delay enable/disable between transactions weight
37 int MONITOR_INSIDE_DELAYEN_WT[] = '{1,1,1,1,1};
38 int MONITOR_INSIDE_DELAYDIS_WT[] = '{3,1,1,1,1};
39
40 // Delay between transactions limits
41 int MONITOR_INSIDE_DELAY_LOW[] = '{0,1,1,0,0};
42 int MONITOR_INSIDE_DELAY_HIGH[] = '{0,3,3,0,3};
43
44 // -------- TEST PARAMETERS ------------------------
45 // Count of transactions to generate
46 parameter TRANSACTION_COUNT = 20_000;
```

Výpis H.2: Konzolový výstup verifikácie

```
1  # ################################# #
2  #    P4 Pipeline verificaniton     #
3  # ################################# #
4
5  # ######### TEST CASE 0 ############
6  # ----------------------------------
7  # --  TRANSACTION TABLE
8  # ----------------------------------
9  # Size:           0
10 # Items added:       20000
11 # Items removed:       20000
12 # ----------------------------------
13
14 # ######### TEST CASE 1 ############
15 # ----------------------------------
16 # --  TRANSACTION TABLE
17 # ----------------------------------
18 # Size:           0
19 # Items added:       20000
20 # Items removed:       20000
21 # ----------------------------------
22
23 # ######### TEST CASE 2 ############
24 # ----------------------------------
```

```
25 # --  TRANSACTION TABLE
26 # ---------------------------------
27 # Size:           0
28 # Items added:       20000
29 # Items removed:       20000
30 # ---------------------------------
31
32 # ######### TEST CASE 3 #############
33 # ---------------------------------
34 # --  TRANSACTION TABLE
35 # ---------------------------------
36 # Size:           0
37 # Items added:       20000
38 # Items removed:       20000
39 # ---------------------------------
40
41 # ######### TEST CASE 4 #############
42 # ---------------------------------
43 # --  TRANSACTION TABLE
44 # ---------------------------------
45 # Size:           0
46 # Items added:       20000
47 # Items removed:       20000
48 # ---------------------------------
```

I Úprava kompilátoru

Výpis I.1: Popis VHDL entity kontrolného bloku MyComputeChecksum

```
1  // ComputeChecksum control block top entity template
2  const std::string compute_checksum_top_entity_template =/
3  R"DELIM(
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  use work.math_pack.all;
9
10 entity {{ entity_name }} is
11
12 port (
13   -- Common interface
14   CLK                     : in std_logic;
15   RESET                   : in std_logic;
16
17   -- ----------------------------------------
18   -- Header interface (input)
19   -- ----------------------------------------
20   {{ header_in_interface.complete_declaration }}
21
22   INPUT_DISCARD           : in std_logic;
23   INPUT_EMPTY_PAYLOAD     : in std_logic;
24   INPUT_READY             : in std_logic;
25   INPUT_NEXT              : out std_logic;
26
27   -- ----------------------------------------
28   -- Metadata interface (input)
29   -- ----------------------------------------
30   -- TODO: metadata input interface
31
32   -- ----------------------------------------
33   -- Payload interface (input)
34   -- ----------------------------------------
35
```

```
36  INPUT_PAYLOAD_DATA    : in std_logic_vector(((/
37  {{ regions }} * {{ region_size }} * {{ block_size }} * /
38  {{ item_width }}) - 1) downto 0);
39  INPUT_PAYLOAD_SOF     : in std_logic_vector(((/
40  {{ regions }} * {{ region_size }}) - 1) downto 0);
41  INPUT_PAYLOAD_SOF_POS : in std_logic_vector(((/
42  {{ regions }} * {{ region_size }} * /
43  max(1, log2({{ block_size }}))) - 1) downto 0);
44  INPUT_PAYLOAD_EOF     : in std_logic_vector(((/
45  {{ regions }} * {{ region_size }}) - 1) downto 0);
46  INPUT_PAYLOAD_EOF_POS : in std_logic_vector(((/
47  {{ regions }} * {{ region_size }} * /
48  max(1, log2({{ block_size }} * 8))) - 1) downto 0);
49  INPUT_PAYLOAD_SRC_RDY : in std_logic;
50  INPUT_PAYLOAD_DST_RDY : out std_logic;
51
52  -- -----------------------------------------
53  -- Header interface (output)
54  -- -----------------------------------------
55  {{ header_out_interface.complete_declaration }}
56
57  OUTPUT_DISCARD       : out std_logic;
58  OUTPUT_EMPTY_PAYLOAD : out std_logic;
59  OUTPUT_READY         : out std_logic;
60  OUTPUT_NEXT          : in std_logic;
61
62  -- -----------------------------------------
63  -- Metadata interface (output)
64  -- -----------------------------------------
65  -- TODO: metadata output interface
66
67  -- -----------------------------------------
68  -- Payload interface (output)
69  -- -----------------------------------------
70  OUTPUT_PAYLOAD_DATA    : out std_logic_vector(((/
71  {{ regions }} * {{ region_size }} * {{ block_size }} * /
72  {{ item_width }}) - 1) downto 0);
73  OUTPUT_PAYLOAD_SOF     : out std_logic_vector(((/
74  {{ regions }} * {{ region_size }}) - 1) downto 0);
```

```
75    OUTPUT_PAYLOAD_SOF_POS : out std_logic_vector(((/
76    {{ regions }} * {{ region_size }} * /
77    max(1, log2({{ block_size }}))) - 1) downto 0);
78    OUTPUT_PAYLOAD_EOF     : out std_logic_vector(((/
79    {{ regions }} * {{ region_size }}) - 1) downto 0);
80    OUTPUT_PAYLOAD_EOF_POS : out std_logic_vector(((/
81    {{ regions }} * {{ region_size }} * /
82    max(1, log2({{ block_size }} * 8))) - 1) downto 0);
83    OUTPUT_PAYLOAD_SRC_RDY : out std_logic;
84    OUTPUT_PAYLOAD_DST_RDY : in std_logic;
85
86    -- ----------------------------------------
87    -- MI32 Interface
88    -- ----------------------------------------
89    DWR  : in std_logic_vector((32 - 1) downto 0);
90    ADDR : in std_logic_vector((32 - 1) downto 0);
91    BE   : in std_logic_vector((4 - 1) downto 0);
92    RD   : in std_logic;
93    WR   : in std_logic;
94    ARDY : out std_logic;
95    DRD  : out std_logic_vector((32 - 1) downto 0);
96    DRDY : out std_logic
97  );
98
99  end entity;
100 )DELIM";
```

Výpis I.2: Popis VHDL prázdnej architektúry kontrolného bloku MyComputeChecksum

```
1  // ComputeChecksum control block top empty architecture
2  // template
3  const std::string /
4  compute_checksum_top_architecture_empty_template =/
5  R"DELIM(
6  library ieee;
7  use ieee.std_logic_1164.all;
8  use ieee.numeric_std.all;
9
```

```
10 use work.math_pack.all;
11
12 architecture empty of {{ entity_name }} is
13
14 begin
15
16   -- HEADERS interface
17   {% for port in hdr_ports %}
18   {{port.left}} <= {{port.right}};
19   {% endfor %}
20
21   OUTPUT_DISCARD <= INPUT_DISCARD;
22   OUTPUT_EMPTY_PAYLOAD <= INPUT_EMPTY_PAYLOAD;
23   OUTPUT_READY <= INPUT_READY;
24   INPUT_NEXT <= OUTPUT_NEXT;
25
26   -- PAYLOAD interface
27   OUTPUT_PAYLOAD_DATA     <= INPUT_PAYLOAD_DATA;
28   OUTPUT_PAYLOAD_SOF      <= INPUT_PAYLOAD_SOF;
29   OUTPUT_PAYLOAD_SOF_POS  <= INPUT_PAYLOAD_SOF_POS;
30   OUTPUT_PAYLOAD_EOF      <= INPUT_PAYLOAD_EOF;
31   OUTPUT_PAYLOAD_EOF_POS  <= INPUT_PAYLOAD_EOF_POS;
32   OUTPUT_PAYLOAD_SRC_RDY  <= INPUT_PAYLOAD_SRC_RDY;
33   INPUT_PAYLOAD_DST_RDY   <= OUTPUT_PAYLOAD_DST_RDY;
34
35   -- MI32 interface
36   DRD                 <= X"DEADBEEF";
37   ARDY                <= WR or RD;
38   DRDY                <= RD;
39
40 end architecture;
41 )DELIM";
```

Výpis I.3: Rozdiel úpravy predlohy top_architecture_full_template

```
1   * @file top_template.h
2 @@ -149,6 +125,27 @@ architecture full of /
3 {{ entity_name }} is
4   signal pd_src_rdy : std_logic;
```

```
5    signal pd_dst_rdy : std_logic;
6    signal pd_afull   : std_logic;
7  + -- Payload checksum input
8  + signal input_cc_pd_data : std_logic_vector({{regions}}/
9    *{{region_size}}*{{block_size}}*{{item_width}}-1 /
10   downto 0);
11 + signal input_cc_pd_sof_pos : std_logic_vector(/
12   {{regions}}*{{region_size}}*max(1,/
13   log2({{block_size}}))-1 downto 0);
14 + signal input_cc_pd_eof_pos : std_logic_vector(/
15   {{regions}}*{{region_size}}*max(1,/
16   log2({{block_size}}*8))-1 downto 0);
17 + signal input_cc_pd_sof : std_logic_vector(/
18   {regions}}*{{region_size}}-1 downto 0);
19 + signal input_cc_pd_eof : std_logic_vector(/
20   {{regions}}*{{region_size}}-1 downto 0);
21 + signal input_cc_pd_src_rdy : std_logic;
22 + signal input_cc_pd_dst_rdy : std_logic;
23 + -- Payload compute checksum output
24 + signal cc_pd_data : std_logic_vector(/
25   {{regions}}*{{ region_size }}*{{ block_size }}*/
26   {{item_width}}-1 downto 0);
27 + signal cc_pd_sof_pos : std_logic_vector(/
28   {{regions}}*{{region_size}}*max(1,/
29   log2({{ block_size }}))-1 downto 0);
30 + signal cc_pd_eof_pos : std_logic_vector(/
31   {{regions}}*{{region_size}}*max(1,/
32   log2({{ block_size }}*8))-1 downto 0);
33 + signal cc_pd_sof : std_logic_vector({{regions}}*/
34   {{region_size}}-1 downto 0);
35 + signal cc_pd_eof : std_logic_vector({{regions}}*/
36   {{region_size}}-1 downto 0);
37 + signal cc_pd_src_rdy : std_logic;
38 + signal cc_pd_dst_rdy : std_logic;
39 + -- Payload checksum NEXT & READY signals
40 + signal cc_out_discard  : std_logic;
41 + signal cc_out_pd_empty : std_logic;
42 + signal cc_out_ready    : std_logic;
43 + signal cc_out_next     : std_logic;
```

```
44    -- Match+Action signlas
45    signal ma_out_discard  : std_logic;
46    signal ma_out_pd_empty : std_logic;
47
48  + -- TMP interconnection payload fifo -> compute checksum
49  + input_cc_pd_data({{regions}}*{{region_size}}*/
50  {{block_size}}*{{item_width}}-1 downto {{regions}}*/
51  {{region_size}}*64) <= (others => '0');
52  + input_cc_pd_data({{regions}}*{{region_size}}*64-1 /
53    downto 0) <= pd_data;
54  + input_cc_pd_sof(0) <= pd_sop;
55  + -- input_cc_pd_sof_pos <= ;
56  + input_cc_pd_eof(0) <= pd_eop;
57  + input_cc_pd_eof_pos({{regions}}*{{region_size}}*max(1,/
58    log2({{block_size}}*8))-1 downto log2({{regions}}*/
59    {{region_size}}*8)) <= (others => '0');
60  + input_cc_pd_eof_pos(log2({{regions}}*{{region_size}}*/
61    8)-1 downto 0) <= pd_eop_pos;
62  + input_cc_pd_src_rdy <= pd_src_rdy;
63  + pd_dst_rdy <= input_cc_pd_dst_rdy;
64  +
65  + -- compute checksum control block
66  + compute_checksum_cb_i : entity /
67  work.{{compute_checksum_entity_name}}
68  + port map(
69  +   -- Common interface
70  +   CLK          => CLK,
71  +   RESET        => RESET,
72  +
73  +   -- ---------------------------------
74  +   -- Header interface (input)
75  +   -- ---------------------------------
76  +   {% for port in cc_in_hdr_ifc %}
77        {{ port.left }} => {{ port.right }},
78  +   {% endfor %}
79  +
80  +   INPUT_DISCARD           => ma_out_discard,
81  +   INPUT_EMPTY_PAYLOAD     => ma_out_pd_empty,
82  +   INPUT_READY             => ma_out_ready,
```

```
83 +    INPUT_NEXT              => ma_out_next,
84 +
85 +    -- --------------------------------
86 +    -- Metadata interface (input)
87 +    -- --------------------------------
88 +    -- TODO: metadata input interface
89 +
90 +    -- --------------------------------
91 +    -- Payload interface (input)
92 +    -- --------------------------------
93 +    INPUT_PAYLOAD_DATA      => input_cc_pd_data,
94 +    INPUT_PAYLOAD_SOF       => input_cc_pd_sof,
95 +    INPUT_PAYLOAD_SOF_POS   => input_cc_pd_sof_pos,
96 +    INPUT_PAYLOAD_EOF       => input_cc_pd_eof,
97 +    INPUT_PAYLOAD_EOF_POS   => input_cc_pd_eof_pos,
98 +    INPUT_PAYLOAD_SRC_RDY   => input_cc_pd_src_rdy,
99 +    INPUT_PAYLOAD_DST_RDY   => input_cc_pd_dst_rdy,
100 +
101 +    -- --------------------------------
102 +    -- Header interface (output)
103 +    -- --------------------------------
104 +    {% for port in cc_out_hdr_ifc %}
105       {{ port.left }} => {{ port.right }},
106 +    {% endfor %}
107 +
108 +    OUTPUT_DISCARD          => cc_out_discard,
109 +    OUTPUT_EMPTY_PAYLOAD    => cc_out_pd_empty,
110 +    OUTPUT_READY            => cc_out_ready,
111 +    OUTPUT_NEXT             => cc_out_next,
112 +
113 +    -- --------------------------------
114 +    -- Metadata interface (output)
115 +    -- --------------------------------
116 +    -- TODO: metadata output interface
117 +
118 +    -- --------------------------------
119 +    -- Payload interface (output)
120 +    -- --------------------------------
121 +    OUTPUT_PAYLOAD_DATA     => cc_pd_data,
```

```
122 +    OUTPUT_PAYLOAD_SOF      => cc_pd_sof,
123 +    OUTPUT_PAYLOAD_SOF_POS  => cc_pd_sof_pos,
124 +    OUTPUT_PAYLOAD_EOF      => cc_pd_eof,
125 +    OUTPUT_PAYLOAD_EOF_POS  => cc_pd_eof_pos,
126 +    OUTPUT_PAYLOAD_SRC_RDY  => cc_pd_src_rdy,
127 +    OUTPUT_PAYLOAD_DST_RDY  => cc_pd_dst_rdy,
128 +
129 +    -- --------------------------------
130 +    -- MI32 Interface
131 +    -- --------------------------------
132 +    DWR                     => (others => '0'),
133 +    ADDR                    => (others => '0'),
134 +    BE                      => (others => '0'),
135 +    RD                      => '0',
136 +    WR                      => '0',
137 +    ARDY                    => open,
138 +    DRD                     => open,
139 +    DRDY                    => open
140 + );
141 +
142     -- Deparser
143     deparser_i : entity work.deparser_top
144     port map(
145 @@ -277,22 +360,22 @@ begin
146     CLK     => CLK,
147     RESET   => RESET,
148     -- Payload interface
149 -   RX_PAYLOAD_DATA     => pd_data,
150 -   RX_PAYLOAD_SOP      => pd_sop,
151 -   RX_PAYLOAD_EOP      => pd_eop,
152 -   RX_PAYLOAD_EOP_POS  => pd_eop_pos,
153 -   RX_PAYLOAD_SRC_RDY  => pd_src_rdy,
154 -   RX_PAYLOAD_EOP      => pd_eop,
155 -   RX_PAYLOAD_EOP_POS  => pd_eop_pos,
156 -   RX_PAYLOAD_SRC_RDY  => pd_src_rdy,
157 -   RX_PAYLOAD_DST_RDY  => pd_dst_rdy,
158 +   RX_PAYLOAD_DATA     => cc_pd_data({{regions}}*/
159     {{region_size}}*64-1 downto 0),
160 +   RX_PAYLOAD_SOP      => cc_pd_sof(0),
```

```
161 + RX_PAYLOAD_EOP      => cc_pd_eof(0),
162 + RX_PAYLOAD_EOP_POS  => cc_pd_eof_pos(log2({{regions}}*/
163   {{region_size}}*8)-1 downto 0),
164 + RX_PAYLOAD_SRC_RDY  => cc_pd_src_rdy,
165 + RX_PAYLOAD_DST_RDY  => cc_pd_dst_rdy,
166
167   -- Payload meta data (if payload is or is not available)
168 - PAYLOAD_EMPTY  => ma_out_pd_empty,
169 - DISCARD        => ma_out_discard,
170 + PAYLOAD_EMPTY  => cc_out_pd_empty,
171 + DISCARD        => cc_out_discard,
172   -- Header interface (output)
173   {% for port in deparser_hdr_ifc %}
174     {{ port.left }} => {{ port.right }},
175   {% endfor %}
176   -- Next, ready interface
177 - EX_READY => ma_out_ready,
178 - EX_NEXT  => ma_out_next,
179 + EX_READY => cc_out_ready,
180 + EX_NEXT  => cc_out_next,
```

Výpis I.4: Predloha compute_checksum_top_architecture_full_template

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4
 5 use work.math_pack.all;
 6
 7 architecture empty of {{ entity_name }} is
 8
 9   {% if hash_over_payload == 1 %}
10     signal fifo_payload_input_src_rdy : std_logic;
11     signal fifo_payload_input_dst_rdy : std_logic;
12     signal fifo_payload_data : std_logic_vector(/
13     (512 - 1) downto 0);
14     signal fifo_payload_sop : std_logic;
15     signal fifo_payload_eop : std_logic;
16     signal fifo_payload_eop_pos : std_logic_vector(/
```

```
17      (log2(512 / 8) - 1) downto 0);
18      signal fifo_payload_src_rdy : std_logic;
19      signal fifo_payload_dst_rdy : std_logic;
20
21      signal sha3_hash_rx_src_rdy : std_logic;
22      signal sha3_hash_rx_dst_rdy : std_logic;
23
24      signal sha3_hash_data : std_logic_vector(/
25      (512 - 1) downto 0);
26      signal sha3_hash_ok : std_logic;
27      signal sha3_hash_vld : std_logic;
28      signal sha3_hash_dst_rdy : std_logic;
29
30    {% endif %}
31
32  begin
33
34    {% if hash_over_payload == 1 %}
35      {% for port in hdr_ports %}
36        {% if port.left == output_hash_port %}
37          {{port.left}} <= sha3_hash_data((512 - 1) downto /
38          (512 - {{output_hash_port_width}}));
39        {% else %}
40          {{port.left}} <= {{port.right}};
41        {% endif %}
42      {% endfor %}
43
44      OUTPUT_DISCARD <= INPUT_DISCARD;
45      OUTPUT_EMPTY_PAYLOAD <= INPUT_EMPTY_PAYLOAD;
46      OUTPUT_READY <= INPUT_READY and sha3_hash_vld;
47      INPUT_NEXT <= OUTPUT_NEXT and sha3_hash_vld;
48
49      INPUT_PAYLOAD_DST_RDY <= fifo_payload_input_dst_rdy /
50      and sha3_hash_rx_dst_rdy;
51      fifo_payload_input_src_rdy <= INPUT_PAYLOAD_SRC_RDY /
52      and sha3_hash_rx_dst_rdy;
53      sha3_hash_rx_src_rdy <= INPUT_PAYLOAD_SRC_RDY /
54      and fifo_payload_input_dst_rdy;
55
```

```
56      -- Payload FIFO
57      payload_fifo_i:entity work.FLU_FIFO
58      generic map(
59        DATA_WIDTH          => 512,
60        SOP_POS_WIDTH       => 1,
61        USE_BRAMS           => True,
62        ITEMS               => 128,
63        BLOCK_SIZE          => 1,
64        STATUS_WIDTH        => 1,
65        OUTPUT_REG          => True
66      )
67      port map(
68        CLK               => CLK,
69        RESET             => RESET,
70
71        -- Frame Link Unaligned input interface
72        RX_DATA           => INPUT_PAYLOAD_DATA((512 - 1) /
73        downto 0),
74        RX_SOP_POS        => (others => '0'),
75        RX_EOP_POS        => INPUT_PAYLOAD_EOF_POS((log2/
76        (512 / 8) - 1) downto 0),
77        RX_SOP            => INPUT_PAYLOAD_SOF(0),
78        RX_EOP            => INPUT_PAYLOAD_EOF(0),
79        RX_SRC_RDY        => fifo_payload_input_src_rdy,
80        RX_DST_RDY        => fifo_payload_input_dst_rdy,
81
82        -- Frame Link Unaligned output interface
83        TX_DATA           => fifo_payload_data,
84        TX_SOP_POS        => open,
85        TX_EOP_POS        => fifo_payload_eop_pos,
86        TX_SOP            => fifo_payload_sop,
87        TX_EOP            => fifo_payload_eop,
88        TX_SRC_RDY        => fifo_payload_src_rdy,
89        TX_DST_RDY        => fifo_payload_dst_rdy,
90
91        -- FIFO state signals
92        LSTBLK           => open,
93        FULL             => open,
94        EMPTY            => open,
```

```
95        STATUS          => open,
96        FRAME_RDY       => open
97      );
98
99      fifo_payload_dst_rdy <= OUTPUT_PAYLOAD_DST_RDY;
100
101     OUTPUT_PAYLOAD_DATA((512 - 1) downto 0) /
102     <= fifo_payload_data;
103     OUTPUT_PAYLOAD_SOF(0) <= fifo_payload_sop;
104     OUTPUT_PAYLOAD_SOF_POS <= (others => '0');
105     OUTPUT_PAYLOAD_EOF(0) <= fifo_payload_eop;
106     OUTPUT_PAYLOAD_EOF_POS((log2(512 / 8) - 1) downto 0) /
107     <= fifo_payload_eop_pos;
108     OUTPUT_PAYLOAD_SRC_RDY <= fifo_payload_src_rdy;
109
110     flu_wrapper_sha3_i : entity work.FLU_WRAPPER_SHA3
111     port map(
112         CLK             => CLK,
113         RESET           => RESET,
114
115         RX_DATA         => INPUT_PAYLOAD_DATA((512 - 1) /
116         downto 0),
117         RX_SOP          => INPUT_PAYLOAD_SOF(0),
118         RX_SOP_POS      => (others => '0'),
119         RX_EOP          => INPUT_PAYLOAD_EOF(0),
120         RX_EOP_POS      => INPUT_PAYLOAD_EOF_POS((log2/
121         (512 / 8) - 1) downto 0),
122         RX_SRC_RDY      => sha3_hash_rx_src_rdy,
123         RX_DST_RDY      => sha3_hash_rx_dst_rdy,
124
125         HASH_DATA       => sha3_hash_data,
126         HASH_OK         => sha3_hash_ok,
127         HASH_VLD        => sha3_hash_vld,
128         HASH_DST_RDY    => sha3_hash_dst_rdy
129     );
130
131     sha3_hash_dst_rdy <= OUTPUT_NEXT and sha3_hash_vld;
132
133   {% endif %}
```

```
134
135   -- MI32 interface
136   DRD                 <= X"DEADBEEF";
137   ARDY                <= WR or RD;
138   DRDY                <= RD;
139 end architecture;
```

Výpis I.5: Zdrojový súbor compute_checksum_top_gen.cpp

```
1  #include <nlohmann/json.hpp>
2  #include <inja/inja.hpp>
3  #include <vector>
4  #include <map>
5
6  #include "compute_checksum_top_gen.h"
7  #include "compute_checksum_options.h"
8  #include "util/connection_module.h"
9  #include "util/interface.h"
10 #include "util/fs.h"
11
12 #include "templates/compute_checksum_top_template.h"
13 #include "templates/modules_tcl_template.h"
14
15 using namespace compute_checksum;
16
17 ComputeChecksumTopGen::ComputeChecksumTopGen(const /
18 ComputeChecksumOptions& options, const IR::ToplevelBlock /
19 *toplevel, const util::Interface &interface, /
20 util::CryptoExternMap *cryptoExternMap) :
21   m_options(options),
22   m_toplevel(toplevel),
23   m_program(toplevel->getProgram()),
24   m_interface(interface),
25   m_entity_name("compute_checksum_top"),
26   m_cryptoExternMap(cryptoExternMap)
27 {
28   m_supportedCryptoExterns = new /
29   crypto::SupportedCryptoExterns();
30   m_hash_over_payload = 0;
```

```
31    m_interface.RemoveArrayType();
32    checkExterns();
33  }
34
35  void ComputeChecksumTopGen::checkExterns()
36  {
37    if (!m_cryptoExternMap->hasExternsInCryptoBlock/
38    ("MyComputeChecksum")) return;
39    for (util::CryptoExtern ext: m_cryptoExternMap->/
40    getCryptoExterns("MyComputeChecksum")) {
41      if (m_supportedCryptoExterns->/
42      isSupported("MyComputeChecksum", ext.m_type, 1)) {
43        m_hash_over_payload = 1;
44      }
45    }
46  }
47
48  std::string ComputeChecksumTopGen::genEntity() const
49  {
50    nlohmann::json ent_data_json;
51
52    ent_data_json["regions"] = /
53    m_options.m_base_options.m_Regions;
54    ent_data_json["region_size"] = /
55    m_options.m_base_options.m_RegionSize;
56    ent_data_json["block_size"] = /
57    m_options.m_base_options.m_BlockSize;
58    ent_data_json["item_width"] = /
59    m_options.m_base_options.m_ItemSize;
60    ent_data_json["entity_name"] = /
61    m_entity_name;
62    ent_data_json["header_in_interface"] = /
63    getInputInterface().toJSON();
64    ent_data_json["header_out_interface"] = /
65    getOutputInterface().toJSON();
66
67    return inja::render/
68    (compute_checksum_top_entity_template,ent_data_json);
69  }
```

```
70
71 std::string ComputeChecksumTopGen::genModulesTcl() const
72 {
73   nlohmann::json modules_tcl__data_json;
74
75   modules_tcl__data_json["top_file_ent_name"] = /
76   m_entity_name + "_ent.vhd";
77   if (m_hash_over_payload == 1) {
78     modules_tcl__data_json["top_file_arch_name"] = /
79     m_entity_name + "_arch_full.vhd";
80   }
81   else {
82     modules_tcl__data_json["top_file_arch_name"] = /
83     m_entity_name + "_arch_empty.vhd";
84   }
85   modules_tcl__data_json["hash_over_payload"] = /
86   m_hash_over_payload;
87
88   return inja::render/
89   (compute_checksum_modules_tcl_template, /
90   modules_tcl__data_json);
91 }
92
93 std::string ComputeChecksumTopGen::genEmptyArch() const
94 {
95   nlohmann::json empty_arch_data_json;
96   util::ConnectionModule conn(getInputInterface(), /
97   getOutputInterface(), "", true, true);
98
99   empty_arch_data_json["entity_name"] = /
100  m_entity_name;
101  empty_arch_data_json["hdr_ports"] = /
102  conn.GetRightConnection();
103
104  return inja::render/
105  (compute_checksum_top_architecture_empty_template, /
106  empty_arch_data_json);
107 }
108
```

```
109 std::string ComputeChecksumTopGen::genFullArch() const
110 {
111   nlohmann::json full_arch_data_json;
112   util::ConnectionModule conn(getInputInterface(), /
113   getOutputInterface(), "", true, true);
114 
115   full_arch_data_json["entity_name"] = /
116   m_entity_name;
117   full_arch_data_json["output_hash_port"] = /
118   "OUT_EX_ETHERNET_DSTADDR";
119   full_arch_data_json["output_hash_port_width"] = 48;
120   full_arch_data_json["hash_over_payload"] = /
121   m_hash_over_payload;
122   full_arch_data_json["hdr_ports"] = /
123   conn.GetRightConnection();
124 
125   return inja::render/
126   (compute_checksum_top_architecture_full_template, /
127   full_arch_data_json);
128 }
129 
130 void ComputeChecksumTopGen::generateVhdl()
131 {
132   util::FileSystem fs;
133   fs.dir = m_options.m_compute_checksum_top_path;
134   fs.fileType = "vhd";
135   fs.fileName = m_entity_name + "_ent";
136   fs.writeToOut(genEntity());
137   if (m_hash_over_payload == 1) {
138     fs.fileName = m_entity_name + "_arch_full";
139     fs.writeToOut(genFullArch());
140   }
141   else {
142     fs.fileName = m_entity_name + "_arch_empty";
143     fs.writeToOut(genEmptyArch());
144   }
145 }
146 
147 void ComputeChecksumTopGen::generateModulesTcl()
```

```
148 {
149   util::FileSystem fs;
150   fs.dir = m_options.m_compute_checksum_top_path;
151   fs.fileType = "tcl";
152   fs.fileName = "Modules";
153   fs.writeToOut(genModulesTcl());
154 }
155
156 const util::Interface ComputeChecksumTopGen::/
157 getInputInterface() const
158 {
159   return m_interface.DeepCopy("IN_", "in");
160 }
161
162 const util::Interface ComputeChecksumTopGen::/
163 getOutputInterface() const
164 {
165   return m_interface.DeepCopy("OUT_", "out");
166 }
167
168 std::string ComputeChecksumTopGen::getTopName() const
169 {
170   return std::string(m_entity_name);
171 }
```